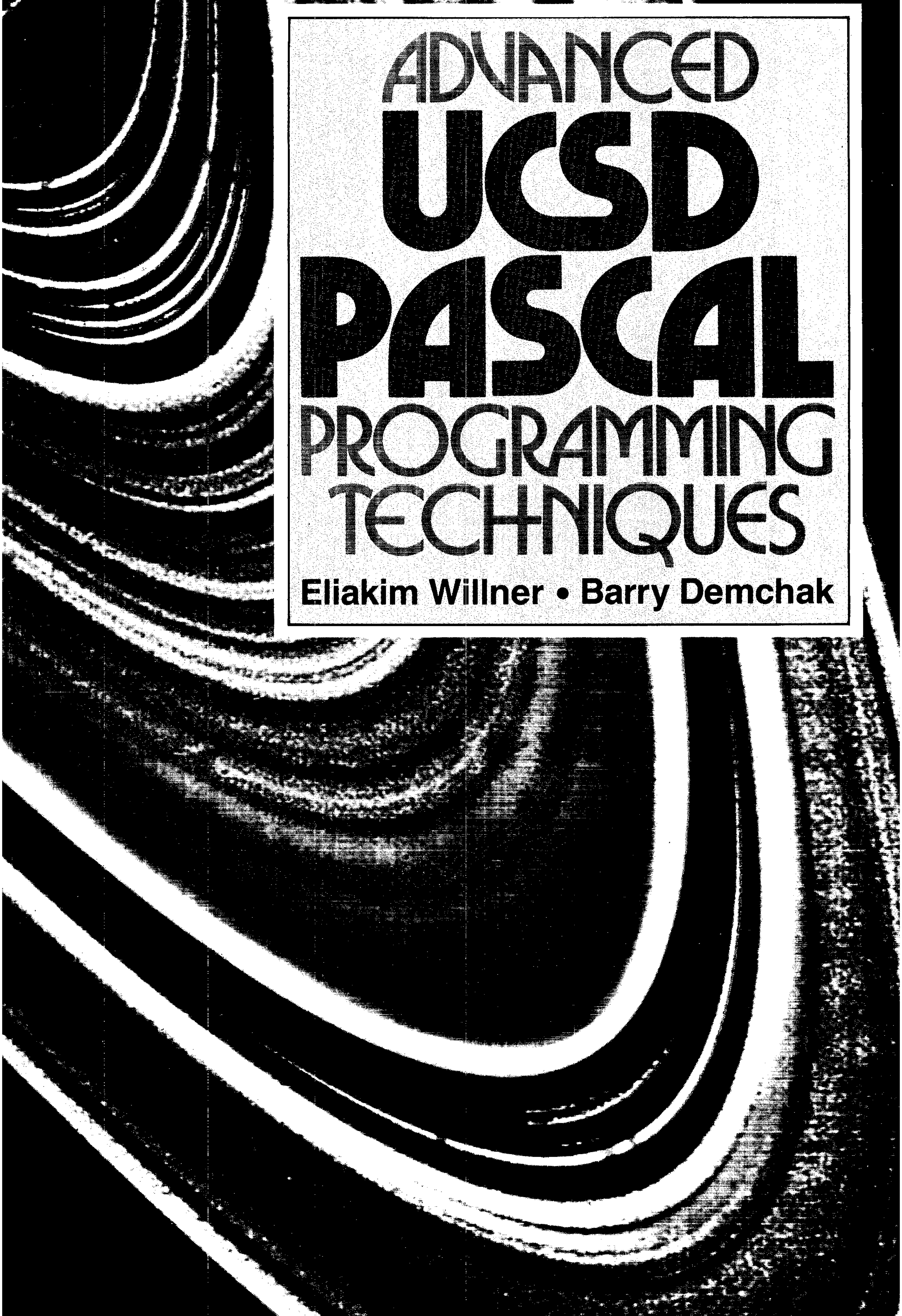
ADVANCED UCSD PASCAL PROGRAMMING TECHNIQUES

Eliakim Willner • Barry Demchak



Advanced UCSD Pascal

Programming Techniques

Eliakim Willner, Datronics, Inc.
Barry Demchak, Software Construction, Inc.

Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data

Willner, Eliakim. (date)
Advanced UCSD Pascal programming techniques.

Includes index.
1. UCSD Pascal (Computer program language)
I. Demchak, Barry. II Title.
QA76.73.U25W55 1985 001.64'24 85-3451
ISBN 0-13-011610-6

Editorial production supervision: Karen Skrable Fortgang
Composed by: Scenic Computer Systems Corporation/ScenicWriter
Manufacturing buyer: Gordon Osbourne

Apple is a registered trademark of Apple Computer, Inc.
Apple Pascal is a trademark of Apple Computer, Inc.
p—System T.M. is a trademark of SofTech Microsystems, Inc.
QBus is a registered trademark of Digital Equipment Computer Corp.
UCSD p—System is a trademark of the
Regents of the University of California.
UCSD Pascal® is a registered trademark of the
Regents of the University of California.

© Copyright 1985 by Advanced Digital Products

The authors and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentatin contained in this book. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-011610-6 01

Prentice—Hall International, Inc., *London*
Prentice—Hall of Australia Pty. Limited, *Sydney*
Editora Prentice—Hall do Brasil, Ltda., *Rio de Janeiro*
Prentice—Hall Canada Inc., *Toronto*
Prentice—Hall Hispanoamericana, S.A., *Mexico*
Prentice—Hall of India Private Limited, *New Delhi*
Prentice—Hall of Japan, Inc., *Tokyo*
Prentice—Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*

Dedication

To my near and extended families. E.G.W.

To my parents, Paul and Sally, who taught me to work hard and always do my best. B.D.

To Roger Sumner, who wrote much of the Pascal System even as it stands today, but who is seldom recognized.

Contents

Foreword xiii
Acknowledgements xv

Chapter 1 **INTRODUCTION** **1**
1.0 Purpose and Scope of this Book 1
1.1 History of UCSD Pascal 2
1.2 Overview 4
1.3 Notation and Terminology 4

Chapter 2 **DEVIATIONS FROM STANDARD PASCAL** **6**
2.0 CASE Statements 7
2.1 GOTO Statements 8
2.2 DISPOSE 9
2.3 NIL 9
2.4 FORWARD 9
2.5 ODD, CHR and NOT 10
2.6 I/O Intrinsics 12
2.6.1 INPUT 12
2.6.2 RESET and REWRITE 12
2.6.3 EOF 12
2.6.4 READ and READLN 13
2.6.5 WRITE and WRITELN 13

2.7 Packed Variables 13
2.8 Procedural and Functional Parameters 14
2.9 Program Headings 14
2.10 Records 14
2.11 Files 15
2.12 Reserved Words 15
2.13 Comments 15
2.14 Type Compatibility 16
2.15 Sets 18
2.16 Pointers 18
2.17 MAXINT 19

Chapter 3 **EXTENSIONS TO STANDARD PASCAL** 20

3.0 Concurrency 23
- 3.0.1 Tasks 24
- 3.0.2 Semaphores 30
- 3.0.3 Interrupts 35
- 3.0.4 Time Slicing 37

3.1 Program Segmentation 38
- 3.1.1 Alternate Segment Management Strategies 42
- 3.1.2 Segments and Tasks 44

3.2 Separate Compilation 44
- 3.2.1 Units 45
- 3.2.2 Using Units 50
- 3.2.3 Unit Linkage 52

3.3 Files 54
- 3.3.1 File System Access 55
- 3.3.2 Interactive Files 58
- 3.3.3 The Keyboard File 59
- 3.3.4 Block Files 60
- 3.3.5 Random Access Files 61

3.4 Strings 63
- 3.4.1 String Parameters 66

3.5 Dynamic Variable Management 68
- 3.5.1 The Version II Heap 69
- 3.5.2 The Version IV Heap 72

3.6 Extended Precision Arithmetic 74
- 3.6.1 Long Integer Parameters 76

3.7 Extended Comparisons 77
- 3.7.1 Records and Arrays 77
- 3.7.2 Pointers 79

3.8 Byte Array Manipulation 79
3.9 Device I/O 84

3.9.1 UNITREAD and UNITWRITE 84
3.9.2 UNITCLEAR, UNITBUSY and UNITWAIT 87
3.9.3 UNITSTATUS 88
3.10 Inline Machine Code 89
3.11 Miscellaneous Extensions 91
3.11.1 Identifiers 92
3.11.2 Declaration Parts 92
3.11.3 Pointer Type Conversion and Comparison 94
3.11.4 Screen Control 94
3.11.5 Clock Access 95
3.11.6 Powers of Ten 97
3.11.7 Arctangent Synonym 98
3.11.8 Procedure Termination 98
3.11.9 I/O Completion Status 100
3.11.10 Memory Available 102
3.11.11 Programmed Halt 103
3.11.12 Compiler Support - TREESEARCH 103
3.11.13 Compiler Support - IDSEARCH 106
3.11.14 FOR Control Variables 109

Chapter 4 **UCSD INTRINSICS** **111**

4.0 ATAN 113
4.1 ATTACH 114
4.2 BLOCKREAD 114
4.3 BLOCKWRITE 115
4.4 CHAIN 116
4.5 CLOSE 117
4.6 CONCAT 118
4.7 COPY 118
4.8 DELETE 118
4.9 EXCEPTION 119
4.10 EXIT 119
4.11 FILLCHAR 120
4.12 GOTOXY 121
4.13 HALT 121
4.14 IDSEARCH 122
4.15 INSERT 122
4.16 IORESULT 122
4.17 LENGTH 123
4.18 MARK 123
4.19 MEMAVAIL 124
4.20 MEMLOCK 124
4.21 MEMSWAP 125

4.22 MOVELEFT 125
4.23 MOVERIGHT 126
4.24 OPENNEW 127
4.25 OPENOLD 127
4.26 P_MACHINE 127
4.27 POS 128
4.28 PWROFTEN 128
4.29 REDIRECT 129
4.30 RELEASE 129
4.31 RESET 130
4.32 REWRITE 131
4.33 SCAN 131
4.34 SEEK 132
4.35 SEMINIT 133
4.36 SIGNAL 133
4.37 SIZEOF 133
4.38 START 134
4.39 STR 134
4.40 TIME 135
4.41 TREESEARCH 135
4.42 UNITBUSY 136
4.43 UNITCLEAR 136
4.44 UNITREAD 136
4.45 UNITSTATUS 137
4.46 UNITWAIT 138
4.47 UNITWRITE 138
4.48 VARAVAIL 139
4.49 VARDISPOSE 139
4.50 VARNEW 140
4.51 WAIT 140

Chapter 5 **COMPILE OPTIONS** **141**

5.0 Options 142
5.0.1 Compiled Listings 145
5.0.2 Include Files 147
5.0.3 Using Units 148
5.0.4 Swapping Compiler 149
5.0.5 Conditional Compilation 150
5.0.6 I/O Checks 151
5.0.7 Range Checks 152
5.0.8 GOTO Restriction 153
5.0.9 Copyright Notices 153
5.0.10 Console Display Suppression 154

5.0.11 Segment Residency 154
5.0.12 System Programs 155
5.0.13 Native Code 159
5.0.14 Real Size 160
5.0.15 Symbolic Debugging 161
5.0.16 Byte Sex Flipping 161
5.0.17 Executable Units 162
5.0.18 "Tiny" Compiler 165
5.1 Option Summary 165

Chapter 6 **PROGRAMMING PRACTICES** **168**

6.0 Packed Variables and Storage Allocation 170
6.0.1 Packed Arrays 171
6.0.2 Packed Records 173
6.0.3 Packing and Storage Allocation Rules 174
6.1 Accessing Bytes, Bits and Bit Fields 178
6.1.1 Words 178
6.1.2 Bit Fields 179
6.1.3 Bits 180
6.2 Unsigned Integer Manipulation 182
6.3 Full-word Logical Operations 186
6.4 Variable-sized Array Allocation 186
6.4.1 Version II Heap Strategy 187
6.4.2 Version IV Heap Strategy 189
6.5 Segment and External Procedures in Unit Interfaces 191
6.6 Structured Parameters Using Pointers 192
6.6.1 Technique 193
6.6.2 Heap Management 195
6.7 Passing "Untyped" Parameters 196
6.8 Variant Record Buffer Overlay 201
6.9 Data Prompts 203
6.9.1 Character Prompts 204
6.9.2 Integer Prompts 204
6.9.3 File Prompt 205
6.9.4 Real Prompts 206
6.10 Device Drivers 210
6.10.1 Driver Interface 211
6.10.2 Device Access 212
6.11 Locating Execution Errors 215
6.11.1 Using Compiled Listings 215
6.11.2 Without Using Compiled Listings 216
6.11.3 Further Investigations 217
6.12 Programming with Units 217

6.12.1 Unit Development 217
6.12.2 Using Pre-existing Units 219
6.13 Using Native Code 221
6.13.1 Automated Native Code Generation 221
6.13.2 User-Supplied Native Code 223
6.14 Passing Parameters Between Programs 223
6.15 Coding Style and Optimizations 226
6.15.1 Expressions and Array Indices 226
6.15.2 Multiword Constants 227
6.15.3 Packed Field References 228
6.15.4 Reals and Long Integers 230
6.15.5 Short Forms 231
6.15.6 WITH Statements 234
6.15.7 String Manipulation 236
6.15.8 CASE Statements 238
6.15.9 GOTO Statements 239
6.15.10 Procedure Calls 240
6.15.11 Parameters to Procedures 241

Chapter 7 THE UCSD P-SYSTEM FILE SYSTEM 243

7.0 File System 244
7.0.1 General Overview 244
7.0.2 Syntax Overview 245
7.1 Physical Units 245
7.1.1 Syntax Overview 246
7.1.2 I/O Devices 246
7.2 Logical Volumes 247
7.2.1 Syntax Overview 248
7.2.2 Block-structured (Disk) Volumes 249
7.2.3 Disk Volume Usage 249
7.2.4 System Volumes 250
7.2.5 Prefixed Volumes 250
7.2.6 Disk Directories 251
7.3 Disk Files 252
7.3.1 Syntax Overview 252
7.3.2 File Attributes 252
7.3.3 File Suffixes 254
7.3.4 File Titles 255
7.3.5 File Length and File Length Specifiers 258
7.4 Syntax Specification 259
7.5 Subsidiary Volumes 260
7.5.1 Creating and Initializing Subsidiary Volumes .. 260
7.5.2 Restrictions 261

7.6 File Conventions and Applications 262
7.6.1 File Name Prompt Conventions 262
7.6.2 Input Prompts 262
7.6.3 Output Prompts 263
7.6.4 File Access from User Programs 264

Chapter 8 **SYSTEM UNITS AND DATA STRUCTURES** 265
8.0 TextFile Format 267
8.0.1 File Structure 268
8.0.2 Header 269
8.1 Using SCREENOPS Data Structures and Procedures . 270
8.2 COMMANDIO Monitor and I/O Redirection 277
8.3 The KERNEL 280
8.3.1 System Constants 281
8.3.2 Accessing the System Date 281
8.3.3 Using Directories from Programs 282
8.4 Segment Code Management 288
8.4.1 Introduction and Overview 289
8.4.2 Code File Structure 293
8.4.3 Environment Records & Segment Information .. 301
8.4.4 As a Program Runs 305
8.5 File Information Blocks (FIBs) 306
8.6 Accessing Internal Operating System Procedures 309
8.7 The Compiler/Operating System Interface 319

Appendix A **STANDARD I/O RESULTS** 325

Appendix B **STANDARD EXECUTION ERRORS** 327

Appendix C **CONDITIONS CAUSING I/O ERRORS** 329

Appendix D **STANDARD I/O UNIT ATTRIBUTES** 332
D.1 Serial Unit Attributes 333
D.2 Serial Input Attributes 333
D.3 Serial Output Attributes 335
D.4 Block-structured Unit Attributes 335
D.5 I/O Unit Specification 336

Appendix E **RESERVED WORDS** 340

Appendix F **PREDECLARED IDENTIFIERS** 341

Appendix G **IMPLEMENTATION LIMITS** **343**
G.1 Quantitative Limits 343
G.2 Sets 344
G.3 Mixed Expression Evaluation 344
G.4 NIL Pointer References 345
G.5 Record Variant Accesses 345
G.6 FOR Statements 345
G.7 Special Symbols 345
G.8 MOD and DIV with Negative Arguments 345

Appendix H **COMPILER SYNTAX ERRORS** **346**

Appendix I **ASCII CHARACTER SET** **351**

Appendix J **DIFFERENCES BETWEEN UCSD VERSIONS** **352**
J.1 Concurrency 352
J.2 NOT 352
J.3 Long Integers 353
J.4 Transcendental Functions 353
J.5 Segments 353
J.6 Units 353
J.7 TREESEARCH 353
J.8 Intrinsics 353
J.9 I/O Redirection 354
J.10 Pointer Comparison 354
J.11 Procedure Size 354

Bibliography 355

Biography 359

Index 361

Foreword

This is a book for "serious" users of the UCSD Pascal System, as most widely distributed on many different personal computers. It is a book that I wish had been available five years ago.

Niklaus Wirth designed Pascal to be a teaching language–one that would be an expression of the systematic or "structured" methods of writing better programs. Like many others, we at UCSD found Pascal to be too good to be limited to teaching only. We saw it as a superb vehicle for creating large, complex system programs–not just the UCSD p-System itself, but a wide variety of large and complex application programs and products. But to use Wirth's Pascal for these applications in practice on a microcomputer required subtle changes and extensions in Wirth's teaching language Pascal. Thus was born the UCSD Pascal language.

One of the most important aims of the UCSD Pascal Project was to demonstrate that it is highly practical to implement a software system that permits large/complex application programs to run *unaltered* on a wide variety of dissimilar machines. This aim led to the the UCSD Pascal Software System, now expanded and marketed as the UCSD p-System by SofTech Microsystems.

Both the language extensions and the system portability features resulted in a practical software development system inevitably characterized by a rich lore that must be understood for the system to be used effectively. The application program developer who ignores this lore risks creating programs that run factors of 10 slower than the p-System permits, or suffers not implementing important features in his product which indeed are very simple to accomplish.

In writing this book, Barry Demchak and Eli Willner have provided a highly readable compendium of the essential lore needed by any serious user of the p-System. As both a collection of suggested techniques and a reference work, a short study of this book will often save large amounts of time for both professional programmers and advanced students of computer science who use the p-System.

Barry Demchak was a member of the student team which implemented UCSD Pascal up through Versions II.1 and III.0. I congratulate him and his coauthor for a fine addition to the literature of the p-System.

Ken Bowles
Professor (retired) University of California, San Diego

Acknowledgements

Acknowledgements are difficult; the number of people and institutions that have influenced us in the course of writing this book is great. It would be impossible to enumerate all of them but several stand out.

First, we would like to acknowledge Neil Williams and Jim Condit of Advanced Computer Design and Rich Gleaves, formerly of Advanced Computer Design. Neil and Jim provided the environment wherein Rich wrote the original manual set for ACD's PDQ-3 computer. These manuals were comprehensive and well-written enough to serve as an excellent foundation for this book.

In addition, we must also express our enormous gratitude to Professor Kenneth Bowles, not only for graciously accepting our invitation to write the foreword to this book, but also for having the courage and insight that made UCSD Pascal the success it has been over the years.

We would also like to thank our reviewers, who made valuable comments in the later stages of this book.

I (Eliakim) would like to thank the many individuals who contributed to my knowledge of UCSD Pascal. At the risk of sounding like an Academy Awards winner, I would like to thank Randy Bush, Bob Peterson, Arley Dealey, Steve Brecher, Jai Khalsa, Jon Bondy, Ted Powell, Stephen Pickett, Derek Jones, George Schreyer, Monty Solomon, Charles Rockwell and Martin Burger. Of course, my co-author, Barry Demchak, figures prominently in this group.

Erik Smith, of Scenic Computer Systems Corporation, is to be thanked not only for contributing to my UCSD Pascal education but also for the excellent job he did in the production of this book.

I became acquainted with all these individuals through USUS, the UCSD Pascal Users' Society. Programming a complex application without the resource of other experienced users is like trying to single-handedly build a castle in the desert–it's just not worth trying. I urge all UCSD Pascal programmers to become involved in USUS.

I began programming as a student at Brooklyn College. I would like to thank their Computer Science faculty for providing me with a solid background as a student, as well as for providing a fertile ground for growth when I joined them as an adjunct faculty member.

In the same vein I thank colleagues at Datronics, Inc. and at Kingsborough Community College (where I am currently a member of the adjunct faculty) for providing a stimulating working environment.

I thank Mario Anello–manager par excellence–and Bill Zolnowski of Con Ed for a number of challenging data-processing assignments, as well as for having the perspicacity to trust their computer expert to make the right decisions.

My thanks to my parents. Finally, my thanks to my wife, Gittle–whose vision enables me to keep in sight the priorities that lie beyond the immediate alligators.

I (Barry) would like to acknowledge all of my family and friends for their support during the long and painful process of creating the predecessors to this book. Special thanks goes to Dr. Jacqueline Byrne and my partners, Bill Franks, Bob Kolodinsky and David Berger, who encouraged me throughout this project and convinced me that it was worth seeing through. I would also like to recognize and thank two outstanding computer scientists, formerly of the University of California, San Diego: Professor Kenneth Bowles and Dr. Richard Sites, both of whom I admire, respect and thank for imparting parts of their attitudes and wisdom to me.

Eliakim Willner
Barry Demchak

Chapter 1

INTRODUCTION

Contents

1.0 Purpose and Scope of this Book 1
1.1 History of UCSD Pascal 2
1.2 Overview 4
1.3 Notation and Terminology 4

1.0 Purpose and Scope of this Book

This book is a tutorial and reference manual for the UCSD Pascal language and operating system. It is written for programmers who have a working knowledge of UCSD Pascal or extensive experience with Standard Pascal. The intention of this book is to provide such advanced programmers with practical knowledge and perspectives that will enable them to write complex UCSD Pascal programs in the most effective and efficient ways possible. Hence, the material presented herein contains not only descriptions of the language, but also programming examples and architectural discussions not available in any other text.

The emphasis of this book is on the two most popular versions of UCSD Pascal. The major emphasis is placed on Version IV, which is distributed by SofTech Microsystems, Inc. and its licensees. Secondary emphasis is placed on Version II, which is distributed by Apple Computer Corporation as Apple Pascal 1.0 and 1.1. The features provided in Apple Pascal are largely subsets of those found in Version IV. Differences between the two versions are discussed where appropriate.

This book does not describe Standard Pascal nor does it present an introduction to UCSD Pascal. It assumes a basic familiarity with both. Information regarding Standard Pascal can be obtained from *American National Standard Pascal Computer Programming Language* (IEEE, 1982). There are a variety of excellent introductory texts dealing with UCSD Pascal, some of which are listed in the bibliography of this book.

1.1 History of UCSD Pascal

UCSD Pascal is a variant of the Pascal programming language developed by Niklaus Wirth at ETH Zurich in the late 1970's. UCSD Pascal was developed by Dr. Kenneth L. Bowles and the students of the "Pascal project" at the University of California in San Diego. In 1975, Dr. Bowles taught the introductory computer programming class at UCSD. At the time, he taught Algol using the campus' Burroughs 6700 mainframe. He chose Pascal as the successor to Algol because of Pascal's simplicity, elegance and suitability for teaching good programming techniques. He hoped to create a microcomputer-based integrated teaching system, thereby providing students with more personalized and expedient instruction.

The first step was to obtain the Wirth P2 Pascal compiler. This compiler translated Pascal to a pseudo code called p-code. The p-code was meant to executeon a hypothetical machine (called the p-machine), and not the B6700. Instead of retargeting the compiler for the B6700, Dr. Bowles chose to write a p-machine emulator (called an interpreter) that ran on the B6700.

In 1976 the compiler was ported to the DEC PDP-11/10 microcomputer under the RT-11 operating system. Because of the interpretive approach taken on the B6700 the port consisted mostly of writing a p-code interpreter for the PDP-11. The vast majority of the compiler was ported without change.

The next step was to create a stand-alone program development system that contained a file handler, a compiler and an operating system. In order to write such a system in Pascal it was necessary to extend the language in the areas of machine-level I/O, dynamic string processing, random access of files and a host of other areas. Version I.0 of the UCSD

Pascal system was produced in 1976. At this point the UCSD Pascal project consisted of Dr. Bowles and approximately five students.

By the summer of 1977 Version I.3 had been produced and the Pascal project began to distribute the system to other educational institutions and to private parties. The Project packaged system code files, utilities, documentation and source on two DEC eight inch diskettes for $200.00.

Not only did the UCSD Pascal system provide an excellent learning environment, but the interpretive approach gave the promise of software that could be transported between different processors without changing the software or the operating environment. A p-code interpreter was written to run on the Intel 8080 microprocessor, and the entire UCSD Pascal system was transported to the 8080 without changing a line of Pascal code! The 8080 version was released in 1977 and utilized the I/O system provided by a CP/M operating system running on the host 8080. This allowed UCSD Pascal to execute on nearly all 8080 systems with little or no additional work.

By the summer of 1978 the UCSD Pascal system was at Version I.4, there were over 40 students on the Pascal project and the Pascal system was being ported to the TI-9900, the Motorola 6800, the Commodore 6502 and the GA-440. A multi-processing version of the p-machine was being developed in conjunction with Western Digital Corporation. Meanwhile, assembly language and modular compilation facilities were added to the system and released in Version I.5.

By early 1979 Apple Computer had begun to distribute UCSD Pascal Version II.1 as Apple Pascal 1.0. This system was essentially identical to UCSD's versions I.5 and II.0, with the exception of some technical adjustments to the p-machine, some bug fixes and the development of Apple's **Intrinsic Unit** scheme of modular compilation. Meanwhile, Western Digital had produced their custom p-machine (called Version III.0) as a microcoded chipset called the MicroEngine. Since the interpreter was microcoded instead of assembly-coded, the Western Digital processor executed UCSD Pascal four to five times faster than other processors.

As a result of the success of UCSD Pascal in the commercial marketplace the University of California ruled that the UCSD Pascal system should be licensed to a commercial enterprise rather than call into question the non-profit status of the entire University of California system. In the summer of 1979 the development and marketing of the UCSD Pascal system was transferred to SofTech Microsystems.

SofTech Microsystems sponsored the development of UCSD Pascal Version IV. Version IV incorporated the modular programming concepts of UCSD Pascal's Version II.0 and Apple Pascal's Version 1.0. It also incorporated the concurrency primitives of Western Digital's Version III.0. SofTech has continued to sponsor enhancements to Version IV, which is currently marketed as the **p-System.** At this writing, SofTech's current release level is Version IV.13. This book concentrates on Version IV.13, but also discusses Apple Pascal where appropriate.

1.2 Overview

This book is organized into eight chapters. **Introduction** presents an overview of UCSD Pascal along with information helpful in using this book. **Deviations** describes the areas in which UCSD Pascal differs from Standard Pascal. **Extensions** describes features available in UCSD Pascal that are not included in Standard Pascal. **UCSD Intrinsics** provides detailed descriptions of all the UCSD Pascal intrinsic routines. The intrinsics are listed in alphabetic order for easy reference. **Compile Options** describes compiler directives which affect either the compiler's operation or the nature of the code produced. **Programming Practices** describes common programming practices in UCSD Pascal. **The UCSD Pascal File System** explains the nature of files, devices and directories and how these may be manipulated from within programs. **System Units and Data Structures** describes how various operating system functions may be invoked from within a program. The **Appendices** include information on I/O device attributes, implementation size limits, differences between UCSD Pascal release versions and a number of helpful tables.

1.3 Notation and Terminology

This section describes the notation and terminology used in this book to describe UCSD Pascal.

When a new language construct is introduced for the first time, it is important to be able to describe in a general way what a valid use of the new construct looks like. In this book, a variant of Backus-Naur form (BNF) is the notation used for describing the form of language constructs. Meta-words are words which represent a class of words; they are delimited by angular brackets ("<" and ">"). Thus, the words "trout", "salmon", and "tuna" are acceptable substitutions for the meta-word "<fish>"; here is an expression describing the substitution:

```
<fish> ::= trout | salmon | tuna
```

The symbol "::=" indicates that the meta-word on the left-hand side may be substituted with an item from the right-hand side. The vertical bar "|" may be read as "or". It separates possible choices for substitution.

The example above indicates that "trout", "salmon", or "tuna" may be substituted for <fish>.

An item enclosed in square brackets may be optionally substituted into a textual expression; for instance, "[micro]computer" represents the text strings "computer" and "microcomputer".

An item enclosed in curly brackets may be substituted zero or more times into a textual expression. The following expression represents responses to jokes possessing varying degrees of humor:

```
<joke response> ::= {Ha}
```

Ha, HaHa, HaHaHaHa – or nothing at all! – are all valid substitutions for <joke response>.

In many instances, the notation described above is used informally to describe the form required by a language construct. Here are some typical examples:

```
START(<process statement> [,<pid> [,<stacksize> [,<priority>]]])

CONCAT(<string> {,<string>})
```

The syntax for Pascal's IF statement is:

```
IF <Boolean expression> THEN <statement> [ELSE <statement>]
```

The following terms are used in the descriptions of UCSD Pascal: **file name, block, block number, unit,** and **unit number. File name** refers to the system's file naming convention. File names are described in chapter 7. **Block** denotes the basic unit of transfer for disk files; a block is defined as 512 bytes of data. **Block** is also defined in Pascal as the set of declarations and statements comprising a program or procedure. The context will make clear which meaning is intended. **Unit** refers either to a separately compilable module or an I/O unit (as described in Appendix D). **Unit number** applies only to I/O units.

Chapter 2

DEVIATIONS FROM STANDARD PASCAL

Contents

2.0 CASE Statements 7
2.1 GOTO Statements 8
2.2 DISPOSE 9
2.3 NIL 9
2.4 FORWARD 9
2.5 ODD, CHR and NOT 10
2.6 I/O Intrinsics 12
2.6.1 INPUT 12
2.6.2 RESET and REWRITE 12
2.6.3 EOF 12
2.6.4 READ and READLN 13
2.6.5 WRITE and WRITELN 13
2.7 Packed Variables 13
2.8 Procedural and Functional Parameters 14
2.9 Program Headings 14
2.10 Records 14
2.11 Files 15
2.12 Reserved Words 15
2.13 Comments 15
2.14 Type Compatibility 16
2.15 Sets 18

2.16 Pointers 18
2.17 MAXINT 19

This section describes the areas where UCSD Pascal deviates from Standard Pascal. Language differences are considered deviations if they meet the following criteria:

- The differences affect compilation or execution of programs written in Standard Pascal. These deviations affect the transportability of Standard Pascal programs *onto* the UCSD Pascal system; they are generally categorized as implementation restrictions.
- The differences subtly alter the Standard Pascal language definition. These deviations affect the transportability of seemingly Standard Pascal programs written in UCSD Pascal *to* other Pascal implementations; they are generally categorized as "features"!

Sections 2.6.5 (WRITE), 2.9 (program headings) and 2.10 (records) describe deviations belonging to both categories. Sections 2.0 (CASE statements), 2.3 (NIL), 2.5 (ODD and CHR), 2.13 (comments), and 2.14 (type compatibility) describe UCSD Pascal "features". The remaining sections describe implementation restrictions.

NOTE: This section describes language deviations only. Implementation-dependent limits are described in Appendix G.

2.0 CASE Statements

In Standard Pascal, the result of a CASE statement is undefined if the case selector contains a value which is not matched by any case label listed in the statement.

In UCSD Pascal, CASE statements are defined to have no effect in this situation; case selection "falls through", and execution continues with the statement following the CASE statement.

Example of CASE statement:

```
program fallthrough;
var ch: char;
begin
  ch := 'b';
  case ch of
    'a': writeln('ch = "a"');
    'c': writeln('ch = "c"');
    end;
    writeln('No errors from case...');
  end {fallthrough}.
```

This program prints only "No errors from case...". The value of the case selector, `ch`, is not matched by a case label within the case statement, so the case statement has no effect. Execution continues at the WRITELN following the case statement. Had the value of `ch` been 'a' or 'c', one of the WRITELN's within the case would have been executed first, followed by the final writeln.

Many implementations of Pascal provide an OTHERWISE clause to the CASE statement. The effect of this clause is to provide an alternative statement which is executed *only* when none of the case labels match the selector.

The effect of the OTHERWISE clause may be simulated in UCSD Pascal by enclosing the CASE statement within an IF statement:

```
program fallthrough;
var ch: char;
begin
  ch := 'b';
  if ch in ['a', 'c']
     then case ch of
            'a': writeln('ch = "a"');
            'c': writeln('ch = "c"');
          end
     else writeln('No matching label in case ...');
end {fallthrough}.
```

Here, "No matching label in case..." is printed. However, if `ch` were either 'a' or 'c' ONLY the appropriate writeln within the case would be executed. The final writeln would be ignored, since it appears in the ELSE portion of the IF statement.

2.1 GOTO Statements

In UCSD Pascal, the scope of labels accessible to GOTO statements is restricted to a single block; thus, out-of-block GOTO's are not allowed.

Pre-Version IV releases of UCSD Pascal restricted the use of GOTO statements to programs that used the $G+ compiler directive. See section 5.0.8 for details.

NOTE: A limited form of out-of-block GOTO is provided with the EXIT intrinsic (see section 3.11.8 for details).

Example of out-of-block GOTO:

```
program outside;
label 1;

  procedure jump;
  begin
    goto 1;
  end {jump};

begin
  jump;
1:
end {outside}.
```

The compiler will complain that the label `1` in the statement `goto 1;` is undeclared because it is not declared within procedure `jump`.

2.2 DISPOSE

The standard procedure DISPOSE is not implemented in versions of UCSD Pascal prior to Version IV. Earlier versions are limited to the UCSD intrinsics MARK and RELEASE for deallocation of dynamically allocated variables (see section 3.5 for details).

2.3 NIL

Standard Pascal defines the symbol NIL as a reserved word. NIL is a predefined identifier in UCSD Pascal. As a practical matter, the only difference is that the compiler will indicate a syntax error if the programmer attempts to redefine a reserved word, but since NIL is treated as a predefined identifier it may be redeclared within the program (although this is not advisable).

2.4 FORWARD

Standard Pascal defines the symbol FORWARD as a directive lacking any meaning outside of a procedure declaration. FORWARD is a reserved word in UCSD Pascal. Hence neither variables, types, constants nor other structures can have the name FORWARD.

2.5 ODD, CHR and NOT

Standard Pascal defines the standard functions ODD, CHR and NOT to return a result whose ordinal value is within the range of the result type. Thus, ODD and NOT are defined to return a BOOLEAN result whose ordinal value is in the range 0..1, and CHR is defined to return a result of type CHAR whose ordinal value is in the range 0..255.

In UCSD Pascal, ODD and CHR perform the required type conversion, but the data itself is not transformed in any way. The effect of these intrinsics is to permit non-boolean and non-character data to be treated as booleans or characters for the purpose of expression evaluation and assignment. For example, ORD(ODD(56)) returns 0 in Standard Pascal since ODD(56) returns the BOOLEAN value FALSE. But in UCSD Pascal ORD(ODD(56)) returns 56. ODD(56) acts as the BOOLEAN FALSE when used in a BOOLEAN context (only the low-order bit is considered) but has an ordinal value of 56. ODD is defined in this manner to allow logical operations on integer types (see section 6.3).

In UCSD Pascal BOOLEANs typically occupy 16 bits, with the value of the BOOLEAN determined by the low-order bit. Although the NOT operator is required to complement only the low-order bit, in fact it complements the entire 16-bit operand. For example, ORD (NOT FALSE) returns 1 in Standard Pascal, but returns -1 in UCSD Pascal. In UCSD Pascal FALSE is represented as a word containing 16 zero bits. NOT FALSE causes the entire word to be complemented, yielding 16 one bits with a two's complement integer value of -1.

WARNING: Under these rules, variables of type BOOLEAN and CHAR may contain values outside of their defined ordinal ranges. BOOLEAN and CHAR comparisons do not work correctly when their arguments possess out-of-range ordinal values, as they are implemented with full-word comparison operators.

```
if odd(56) = odd(58)
   then write('This must be Standard Pascal')
   else write('This must be UCSD Pascal');
```

Since the ODD function returns a word with the same value as its argument, and since the entire word is considered although a BOOLEAN comparison is taking place, `odd(56)` will not be equal to `odd(58)` in UCSD Pascal.

Array indexing using subscripts of types BOOLEAN and CHAR may not behave as expected. For example:

```
var bool: boolean;
    arry: array[boolean] of integer;
```

```
bool:= false;
array[not bool] := 0;
```

The array reference above functions properly in Standard Pascal but produces an out-of-range subscript in UCSD Pascal; the full word value of `not bool` is 16 "one" bits, or -1.

Note that conditional statements in UCSD Pascal ignore all but the low-order bit of a BOOLEAN result and thus are unaffected by this feature. The WRITELN statement of the following example will be executed, as expected.

```
if odd(57)
  then writeln('The low bit was set');
```

ODD example:

```
program bitdiddle;
const highmask = 255;
var num: integer;
begin
  num := 556;
  num := ord(odd(num) and odd(highmask));

{       0000 0010 0010 1100  (decimal 556; num)
    AND 0000 0000 1111 1111  (decimal 255; highmask)
      = 0000 0000 0010 1100  (decimal 44;
                                 new value of num)

     The high byte of num has been masked off.
     Num now contains the integer value 44     }

  num := ord(not odd(num));

  { NOT 0000 0000 0010 1100  (decimal 44; num )
      = 1111 1111 1101 0011  (decimal -45;
                                 new value of num)

  Taking the 1's complement yields -45 }

end {bitdiddle}.
```

This program illustrates how the ORD and ODD functions may be used to perform boolean AND and NOT operations on 16-bit integer values. First the high byte of `num` is cleared, preserving the low byte, by ANDing `num` with `highmask`. The AND operation is defined only on booleans, however. Thus, the ODD function is used on `num` and `highmask` to cause them to be viewed as booleans while maintaining their original bit configurations. The AND operation produces a boolean result which may be assigned to the integer variable `num` by using the ORD function.

The subsequent statement again uses ODD to permit `num` to be treated as a boolean, this time to permit the use of the NOT operator. NOT complements each of the 16 bits of `num`. ORD is used to permit the result to be assigned back to the integer `num`.

2.6 I/O Intrinsics

Sections 2.6.1 through 2.6.3 describe deviations resulting from UCSD Pascal's file I/O environment. Sections 2.6.4 and 2.6.5 describe deviations resulting from implementation restrictions.

2.6.1 INPUT

The predeclared file INPUT is defined as an interactive file in UCSD Pascal. Section 3.3.2 describes interactive files.

Whether or not I/O redirection has been invoked, all data read from the INPUT file is echoed to the console.

2.6.2 RESET and REWRITE

The standard procedures RESET and REWRITE have been altered to provide direct access to the file system (see section 3.3.1 for details). UCSD Pascal does not allow internal (memory resident) files. All files must be mapped into external files. Internal files may be simulated with temporary external files.

Example of an internal file in Standard Pascal:

```
procedure local;
var internal: file of integer;
begin
  rewrite(internal);
  ...
end {local};
```

In UCSD Pascal, EOF is set to FALSE after a file is rewritten. Standard Pascal defines EOF to be true after rewriting a file. The reason for this discrepancy is explained in the following section.

2.6.3 EOF

UCSD Pascal redefines the meaning of the standard function EOF for files that are open for writing. The standard procedure REWRITE initially sets EOF to FALSE; EOF then serves as a physical end-of-file indicator. The standard procedure PUT sets EOF to false after every successfully written record. If PUT attempts to write a record past the end of the space allocated for the disk file (see section 7.3.5), and the file space cannot be extended (for lack of available disk space after the file), EOF becomes true.

NOTE: Attempts to write beyond the physical end of a file result in a system execution error unless I/O checking is disabled. See section 5.0.6 for details on disabling I/O checking.

2.6.4 READ and READLN

The standard procedure READ may not be applied to files other than text files (files of type text, interactive or file of char). READ and READLN accept elements of packed character arrays or strings as arguments. Entire packed character arrays and strings are also acceptable. READ and READLN are redefined when used with interactive files (see section 3.3.2 for details).

2.6.5 WRITE and WRITELN

The standard procedure WRITE may not be applied to files other than text files.

Standard Pascal defines an optional control parameter named **fraction length** for specifying the output format of values of type REAL. The fraction length parameter specifies the number of digits to follow the decimal point in a fixed point representation of the value. If the fraction length specifies more digits than can be represented as significant digits by the underlying floating point implementation, the standard directs the fraction to be padded out with the requisite number of 0's. UCSD Pascal pads out overly long fractional parts with blank characters in place of (nonsignificant) "0" digits.

Note that most implementations of UCSD Pascal do not permit boolean values to be written with WRITE or WRITELN.

2.7 Packed Variables

The standard procedures PACK and UNPACK are not implemented in UCSD Pascal. UCSD Pascal does perform packing of array and record types preceded by the reserved word PACKED. Variables are UNPACKed and rePACKed transparently as needed (e.g., to assign a value to a single element of a packed array). However, unpacking an entire structure must be done an element at a time. Given the following situation, for example:

```
type
  t = record
    n: integer;
    s: string[9];
  end;
var
  a: packed array[0..10] of t;
  b:        array[0..10] of t;
```

the statement `b := a;` compiles correctly but will not necessarily execute correctly for an arbitrary record `t`. The correct unpacking procedure is as follows:

```
for i : = 0 to 10 do
  b[i] := a[i];
```

2.8 Procedural and Functional Parameters

Procedural and functional parameters are not implemented as of Version IV.13 of UCSD Pascal.

2.9 Program Headings

Parameter lists associated with program headings are ignored in UCSD Pascal. The standard files INPUT and OUTPUT are predeclared and opened to the system console by the operating system. Programs gain access to externalfiles with the intrinsics REWRITE, RESET and CLOSE (see section 3.3.1 for details).

NOTE: The INPUT and OUTPUT files may be attached to input and output streams other than the system console by using the I/O redirection options described in section 4.4.

2.10 Records

UCSD Pascal does not allow records to be declared with empty field lists.

UCSD Pascal does not enforce variant part completeness in record declarations. Thus, the case labels need not specify all possible values of the tag field. For example, the following record declaration is legal even though there are not case labels for all possible integers.

```
type
    devrec = record
                s: string;
                case integer of
                  1: (b: boolean);
                  2: (r: real);
             end;
```

2.11 Files

UCSD Pascal does not allow file variables to be declared as part of an array or record. Nor does it allow dynamic allocation of file variables.

2.12 Reserved Words

A number of reserved words have been added to UCSD Pascal. As noted in section 2.4, FORWARD is a reserved word rather than a directive. The following identifiers are reserved words in UCSD Pascal:

```
external
forward
implementation
interface
process
segment
separate
unit
uses
```

2.13 Comments

Standard Pascal defines the symbols "(*" and "*)" as alternative symbols for the comment delimiters "{" and "}" respectively. Thus, comments may begin with "{" and end with "*)", or begin with "(*" and end with "}". Additionally, comments may not be nested in Standard Pascal.

UCSD Pascal treats "(*" and "*)" as separate comment delimiters from the pair "{" and "}". Thus, comments beginning with "{" must end with "}" and comments beginning with "(*" must end with "*)". As a result, comments may be nested by using one pair of delimiters to comment out source code containing comments delimited by the alternative symbols. The compiler does not consider a comment terminated until it finds a delimiter matching the one that began the comment.

Commented sections of UCSD Pascal programs are flagged as such in compiled listings.

Example of comments in UCSD Pascal:

```
program comments;
begin

  (*
  comment out following statements ...

  writeln('I won''t writeln');    { and it won't! }

  don'twriteln('writeln');   { syntactically incorrect }
                             { but no error message, since }
                             { it's a commented-out statement }
  ... end of comment
  *)

end {comments}.
```

2.14 Type Compatibility

In Pascal, variables may not be used in the same expression, nor may the value of one variable be assigned to another, unless the variables are "type compatible".

In Standard Pascal, the rules for type compatibility are referred to as **name compatibility.** In general, variables are name-compatible if one of the following conditions is true:

- The variables are declared with the same type identifier (e.g., `var v1: stuff; v2: stuff;`).
- The variables are in the same identifier list of a single variable declaration (e.g., `var v1, v2: array[char] of integer;`).

In UCSD Pascal, the rules for type compatibility are referred to as **structure compatibility.** Variables are structure-compatible if the data structures implementing their respective types are structurally equivalent, regardless of whether or not their types have the same name. To be structure-compatible:

- Simple types must share the same base type (note that subrange types that share the same base types are compatible with their base type and with each other).
- Sets must have structure-compatible base types.
- Arrays must have structure-compatible base types and index types along with identical array bounds (as of Version IV.13).

- Records must have structure-compatible fields declared in the same order.

  Example of structure-compatible simple types:

```
type
        length = real;
        weight = real;
```

  Example of structure-compatible arrays:

```
type
        t1 = 1..10;
        t2 = 1..10;

        x = array [t1] of integer;
        y = array [t2] of 0..52;
```

  Example of structure-compatible records:

```
type
      polar = record
                radius, angle: real;
                int: integer;
              end;

      cart  = record
                x: real;
                y: real;
                z: 0...2047;
              end;

      duple = record
                s,t: real;
                u:  10..15;
              end;
```

NOTE: Name compatibility implies structure compatibility but the converse is not true. Thus, UCSD Pascal programs utilizing structural type compatibility will not compile on Standard Pascal compilers which enforce name compatibility.

WARNING: Structural equivalence of records can lead to somewhat strange notions of type compatibility. For instance, assume that the value 1.0 is assigned to the field `x` in a record of type `cart`. If the record is assigned to a record of type `duple`, the value 1.0 is contained in the `t` field of `duple` instead of the `s` field! This is a result of the compiler's scheme for allocating storage space for record fields (see section 6.0.2 for details).

UCSD Pascal programmers are urged to exercise caution when utilizing structural type compatibility.

2.15 Sets

Standard Pascal permits sets to be defined over ranges including negative integers. UCSD Pascal restricts sets defined over integers to positive values. Thus, the following declaration is illegal in UCSD Pascal:

```
var s: set of -10..10;
```

WARNING: In UCSD Pascal sets defined over ranges of positive integers are allocated storage as if the range began at zero. For example,

```
var s: set of 1000..1500;
```

is allocated storage as if the range began at zero instead of 1000. This causes an extra 1000 bits to be allocated for the variable `s`.

Standard Pascal prohibits out-of-range values from being added to sets. UCSD Pascal permits out-of-range values to be added to sets, provided the value is within range of the size allocated for the set.

```
program settest;
var
  s: set of 1..3;
begin
  s := [4, 5, 6];
  if (4 in s)
    then writeln('This is UCSD Pascal')
    else writeln('This is Standard Pascal');
end.
```

In the example above, the values 4, 5 and 6 may be added to the set `s` even though it is declared as set of 1..3. In fact, any value in the range 0..15 may be added to `s`; values outside that range will cause an execution-time error since `s` occupies one word.

2.16 Pointers

UCSD Pascal attempts to resolve pointer references immediately as they are encountered in type declarations. Standard Pascal defers resolution of such references until the end of the type declaration. This can cause incompatibilities when pointer types are declared before their referents in procedures. For example:

```
program who;
type x = real;

  procedure within;
  type y = ↑x;
       x = integer;
```

```
    begin end;

  begin
  end.
```

In Standard Pascal, `y` ends up a pointer to an integer, since the declaration of `y` is not resolved until the entire local type declaration ends. In UCSD Pascal, `y` is a pointer to a real, since the declaration of `y` is resolved immediately, before the compiler is aware of the local type `x`.

2.17 MAXINT

In UCSD Pascal, MAXINT should not be used as the termination value of a FOR loop. Doing so erroneously causes the loop control variable to wrap around from MAXINT to -32768, then to -32767, and so on, cycling up to zero, then back up to MAXINT ...

```
  for i:= 1 to MAXINT do writeln ('Get me out of here!');
```

Chapter 3

EXTENSIONS TO STANDARD PASCAL

Contents

3.0 Concurrency 23
3.0.1 Tasks 24
3.0.1.1 Processes 26
3.0.1.2 Task Identifiers 27
3.0.1.3 Task Stacks 28
3.0.1.4 Priority 29
3.0.2 Semaphores 30
3.0.2.1 Mutual Exclusion 32
3.0.2.2 Synchronization 33
3.0.3 Interrupts 35
3.0.4 Time Slicing 37
3.1 Program Segmentation 38
3.1.1 Alternate Segment Management Strategies 42
3.1.2 Segments and Tasks 44
3.2 Separate Compilation 44
3.2.1 Units 45
3.2.2 Using Units 50
3.2.3 Unit Linkage 52
3.3 Files 54
3.3.1 File System Access 55
3.3.2 Interactive Files 58

3.3.3 The Keyboard File 59
3.3.4 Block Files 60
3.3.5 Random Access Files 61
3.4 Strings 63
3.4.1 String Parameters 66
3.5 Dynamic Variable Management 68
3.5.1 The Version II Heap 69
3.5.2 The Version IV Heap 72
3.6 Extended Precision Arithmetic 74
3.6.1 Long Integer Parameters 76
3.7 Extended Comparisons 77
3.7.1 Records and Arrays 77
3.7.2 Pointers 79
3.8 Byte Array Manipulation 79
3.9 Device I/O 84
3.9.1 UNITREAD and UNITWRITE 84
3.9.2 UNITCLEAR, UNITBUSY and UNITWAIT 87
3.9.3 UNITSTATUS 88
3.10 Inline Machine Code 89
3.11 Miscellaneous Extensions 91
3.11.1 Identifiers 92
3.11.2 Declaration Parts 92
3.11.3 Pointer Type Conversion and Comparison 94
3.11.4 Screen Control 94
3.11.5 Clock Access 95
3.11.6 Powers of Ten 97
3.11.7 Arctangent Synonym 98
3.11.8 Procedure Termination 98
3.11.9 I/O Completion Status 100
3.11.10 Memory Available 102
3.11.11 Programmed Halt 103
3.11.12 Compiler Support - TREESEARCH 103
3.11.13 Compiler Support - IDSEARCH 106
3.11.14 FOR Control Variables 109

This chapter describes UCSD extensions to Standard Pascal. The extensions may be divided into three classes with respect to syntax:

- Reserved Words - A handful of reserved words have been added to support segment procedures, units, and processes. Reserved words are listed in Appendix E.

- Predeclared Types and Routines - These extensions may be used in any UCSD Pascal program; unlike reserved word extensions, predeclared identifiers may be redefined in the program. Two examples of predeclared types are STRING and SEMAPHORE. Predeclared procedures and functions are usually called **intrinsics**; they comprise the majority of language extensions in UCSD Pascal. See chapter 4 for detailed descriptions of the UCSD intrinsics.
- Syntax Extensions - Standard Pascal syntax has been modified to accommodate some extensions. The SCAN intrinsic requires a parameter known as a ”partial Boolean expression”. A partial Boolean expression consists of an ”=” or ”<>” operator followed by a character expression (e.g., `= 's'` is a valid partial Boolean expression). The declaration of a block file appears as a file type declaration lacking a base type specification (e.g., `type blockfile = file;` is a valid type declaration). Type declarations for strings and extended precision integers contain subtype specifications which define the type's size attribute (e.g., `type longint = integer[20];`). Variable addresses are obtained in the PMACHINE intrinsic by preceding a variable reference with the ”up-arrow” symbol (e.g., `↑PERSON.NAME[I]`). Finally, many intrinsics accept optional parameters or parameter sequences.

NOTE: Most extensions described in this chapter are recognized by the compiler and are hence part of the UCSD Pascal base language. Another class of extensions is available through the use of library modules, which may be user-written or purchased from a variety of vendors. Available routines include those that allow program chaining, extended directory management, screen control, and other system and user oriented functions. A number of modules included in many recent p-System releases are described in chapter 8.

Sections 3.0 through 3.2 describe the major extensions to Standard Pascal: concurrency, program segmentation, and separate compilation. Sections 3.3 through 3.6 describe other commonly used extensions: files, strings, dynamic variable management, and precision arithmetic. Sections 3.7 through 3.10 present low-level extensions which possess minimal type-checking and are intended primarily for systems use; these should be used only when necessary. Section 3.11 describes the remaining extensions.

3.0 Concurrency

Most conventional programming languages view a program as a description of a single activity, where an "activity" is something that the processor can accomplish by following a sequential flow of instructions, one at a time.

Many programs would benefit, though, if they were treated not as a single sequential activity, but as a number of activities occurring simultaneously. Consider a program written for a home control application. Such a program might have to monitor sensors located at various points around the house to detect unauthorized entries and at the same time monitor room temperatures to control the home heating system.

Each of these separate monitoring activities must occur at the same time. In a conventional language the programmer would have to simulate this concurrency by, perhaps, coding each activity as an individual subroutine and arranging for the computer to "bounce" back and forth quickly from one subroutine to the other. It is more natural, however, to represent the activities as separate tasks within a program and to visualize the computer as working on all tasks at once.

Concurrency is defined as the simultaneous execution of a number of activities. Since most computer systems have a single processor – capable of focusing on just one activity at a time – they simulate concurrency by implementing "virtual machines" on the physical machine. Each of the concurrent activities executes on its own virtual machine, leaving the physical machine responsible for simulating concurrent execution of the virtual machines. Activities that execute on virtual machines are generally referred to as **tasks**.

Concurrency in UCSD Pascal is restricted to concurrent execution of routines declared in a single program. A program may initiate any number of tasks, thus allowing any number of virtual machines to operate "simultaneously". The tasks must finish executing before the program is allowed to terminate.

Note that the current versions of the p-System will not allow two *programs* to execute simultaneously.

On a single processor system concurrency is simulated by sharing the processor among tasks. Processor sharing is accomplished by allowing each task to execute until a programmer-defined event occurs and then switching the processor to another task. The latter action is known as a **task switch.** The task executing on the processor is called the **current task,** while tasks waiting for processor time are called **ready-to-run** tasks. When ready-to-run tasks are resumed they pick up where they had left off prior to being "switched-out". This is accomplished by storing execution states (i.e., processor register values) describing ready-to-run tasks in a system structure known as the **ready queue.** Note that this scheme imposes the burden of task switching on the operating system rather than on the programmer.

NOTE: The simulation of concurrent tasks occurs only as a result of p-code-level interrupts as described in section 3.0.3. The ability to execute tasks concurrently is severely restricted on implementations not providing these interrupts.

It is often necessary for concurrently executing tasks to coordinate their activities. This is the case when the tasks must share some common data or a single system resource. For example, suppose that the security monitoring task and the temperature monitoring task mentioned above both send output to the terminal. If each attempted to write to the screen at the same time an unreadable display would result. The programmer must assure that each of these tasks secures exclusive access to the terminal before writing to it.

Semaphores are special variables used for task synchronization. Semaphores are used both in preventing tasks from executing until an event occurs, and in signalling occurrences of events. Tasks waiting for an event to occur are called **suspended** tasks. Execution states for suspended tasks are stored in a semaphore's **wait queue.**

This section describes concurrency in UCSD Pascal. **Tasks** are described in section 3.0.1. **Semaphores** and applications of task synchronization are described in section 3.0.2. Section 3.0.3 describes **interrupt handling,** in which semaphores enable tasks to respond to processor interrupts. Section 3.0.4 describes **time slicing,** which allows simulation of true concurrent processing on a single processor machine.

NOTE: See section 6.10 for applications of concurrency to the development of device drivers.

3.0.1 Tasks

A task is defined by four attributes: **process, task identifier, stack size,** and **priority.**

The primary attribute of a task is the code it executes. This code is described in special procedure-like constructs called **processes.** Processes are described in section 3.0.1.1. Each task is assigned a unique identifying value when it is created; this value may be retained in a **task identifier** variable to distinguish the newly started task from other tasks in the system. The task identifier mechanism is needed to distinguish between tasks because, although the code being executed (the process) is unique, there may be more than one task concurrently executing the code of the same process. (In the context of our previous example, we may have many sensors to monitor; the code that defines sensor-monitoring would appear in a single process but we would create a task executing that code for each sensor.) Task identifiers are described in section 3.0.1.2. The amount of memory allocated for a new task is determined by the task's **stack size.** Task stacks are described in section 3.0.1.3. The last (but not least) attribute is **priority**; a task's priority value determines its ability to obtain processor time when competing with other tasks. Task priorities are described in section 3.0.1.4.

The system assigns task attributes when a task is initiated. This occurs via the UCSD intrinsic START, which has the following form:

```
START ( <process call> [, <task identifier variable>
                            [, <stacksize expression>
                              [, <priority expression> ]]] );
```

The main parameter to START is a process call; it resembles a procedure call, and may contain parameters passed to the task (e.g., `START(Zip)` or `START(BackgroundLaughter(30))`). In fact, it is useful to visualize STARTing a process as akin to calling a procedure - but not waiting for the procedure to terminate before continuing. Note that starting a single process several times in a program creates a number of tasks executing the same copy of a process's code independently.

The remaining parameters are optional. A task identifier (second parameter to START) must be declared as a variable of type PROCESSID. The stack size parameter (third parameter) consists of an integer-valued expression, and represents the number of words allocated for the task stack space; the default stack size is 200 words. The priority parameter consists of an integer-valued expression in the range -1..255. Values outside this range cause an execution error to occur. The default priority is 128. These parameters are discussed in detail in the following sections.

Tasks terminate execution when they reach the end of their process code. In pre-Version IV.13 releases task stack spaces occupied memory until the parent program finished execution. In Version IV.13 task stack space is recycled automatically.

The system prevents a program from terminating until all of its tasks have terminated.

NOTE: The description of tasks presented here is sufficient for describing the execution of programs containing processes. From the system's point of view, the entire system (which consists of the operating system kernel and the user program) is called the **main task;** the other tasks (including system device drivers and user-defined processes) are known as **subsidiary tasks.** From the processor's point of view, there is no distinction between the main task and subsidiary tasks; they are functionally equivalent.

3.0.1.1 Processes

Processes are declared similarly to procedures; however, the reserved word PROCESS replaces the reserved word PROCEDURE. The syntax description presented below is derived from the formal syntax description for procedures in Appendix D of the *Pascal User Manual and Report*:

```
<process declaration> ::= <process heading> <block>

<process heading> ::=
    PROCESS <identifier> <formal parameter part> ;
```

Processes must be declared in the outer (global) block of a program; they may not be declared within a procedure or another process. START may only be called from the main task; thus, subsidiary tasks cannot create new tasks. Violating this restriction causes an execution error.

WARNING: Tasks are not allocated their own heap space for their dynamic variables. Dynamic variables are always allocated on the system heap. For this reason, using DISPOSE to deallocate dynamic variables within a task is recommended rather than MARK and RELEASE, as MARK and RELEASE may inadvertently remove variables created by other tasks. Sections of the operating system dealing with global resource management (e.g., the file system and heap) are protected from task contention; nevertheless, processes using these resources should do so carefully. Section 3.0.1.3 describes other problems caused by interactions between tasks and the system heap.

NOTE: Variable parameters passed to a process may require an associated semaphore in order to ensure mutually exclusive access to the actual parameter (see section 3.0.2 for more information).

Examples of process declarations:

```
process Zip;
begin
  ...
end;

process BackgroundLaughter (Laughs: integer);
begin
  i := 0;
  j := 0;
```

```
    if Laughs > 0 then
      repeat
        write('ha');
        i := (i + 1) mod Laughs;
        if i = 0
          then begin
            writeln;
            j := j + 1;
          end;
      until j = 100;
  end {BackgroundLaughter};
```

The `BackgroundLaughter` process, like most processes, takes the form of a loop. It vies for CPU time with the other processes (if any) and prints lines containing `Laughs` number of 'ha's. This laughter continues concurrently with whatever the other processes in the program are doing until one hundred lines of 'ha's are printed. The `BackgroundLaughter` process then terminates.

3.0.1.2 Task Identifiers

START assigns each task a unique value, distinguishing it from other tasks. These values may be obtained by specifying a task identifier variable as the task identifier parameter to the START intrinsic; START assigns the value associated with the new task to the variable. Task identifier variables must be declared with the predefined type PROCESSID, and can be used in the same manner as pointer variables (i.e., the only valid operations are assignment and comparison with other task identifier variables).

In the following example, two tasks are created with START; the variables PID1 and PID2 are assigned values identifying the tasks. Because these values are unique, this program writes "Truth" when executed:

```
Program a;

Var PID1,PID2: processid;

  process t;
  begin
    ...
  end;

begin
  start(t, PID1);
  start(t, PID2);
  if PID1 <> PID2 then writeln ('Truth');
end.
```

3.0.1.3 Task Stacks

Each task is allocated an area of memory in which it can execute; because UCSD Pascal programs execute on stack-oriented machines, the memory area is called a **stack space**. A Stack space is used to store parameters and variables, and procedure call information. When a task exhausts its stack space, a "stack overflow" occurs, and the system must be restarted.

NOTE: The main task's stack space is coincident with the system stack, and is limited in size only by the amount of system memory available. Stack spaces for subsidiary tasks are allocated on the system heap by the START intrinsic; hence, they are generally small compared to the main task's stack space. (Note that the main task's stack competes with the system heap for memory, while a subsidiary task's stack space is of fixed size, and is used only by the process code.)

WARNING: Because task stacks are allocated on the system heap, tasks are susceptible to destruction from careless use of MARK and RELEASE since these cause an absolute and process-independent amount of heap to be freed. Dynamic variables allocated before a started task should be deallocated using DISPOSE; never RELEASE the heap below a started task.

NOTE: Stack sizes must be sufficient for the basic needs of a process. The minimum size depends on the version of the p-System begin used; a typical (conservative) minimum might be approximately 32 words plus the number of words used by local variables and parameters. A procedure call uses a minimum of 5 words of space. Whenever possible avoid calling segments and/or procedures with large local data spaces, as they can quickly consume a task's stack space. (If this is unavoidable, the DECODE utility and compiled listings may be used to reveal the sizes of code and data segments in order to determine the amount of stack space required by a task (see your Architecture Guide for more specific information).)

NOTE: The stack space must be large enough to satisfy the stack requirements of any calls to the operating system resulting from the use of Pascal I/O intrinsics. These requirements must be determined by trial-and-error.

Examples of stack size specification:

```
Program a;
Var PID: processid;
    I,J: integer;

  process p1;
  begin
    . . .
  end;

begin
  I := 4;  J := 5;
  start(p1,PID);                  { stack space = 200 }
  start(p1,PID,10);               { stack space = 10  }
  start(p1,PID,(I + J)*100);      { stack space = 900 }
end.
```

NOTE: In UCSD Pascal Version III the task stack is also used to contain the code of segment procedures called by the process. Provision for the sizes of such segments must be made when calculating the stack size. The DISASSEM utility may be used to determine sizes of segments.

3.0.1.4 Priority

Each task is assigned a **task priority** value between 0 and 255. A task's priority determines its ability to obtain the processor when other tasks are ready to run. The processor's task scheduling policy is simple: no task may execute when a higher priority task is ready to run. The system enforces this policy by ordering all tasks in the ready queue by their priority, and by performing a task switch when the task at the head of the ready queue has higher priority than the current task. This implies that tasks in the ready queue at the same priority level as the currently executing task will not run unless and until the currently executing task becomes suspended. This can occur as a result of a WAIT in the user program. The currently executing task can also become suspended as a result of a WAIT in a system routine called by a program –such as when a segment is read into memory, or during a file access.

NOTE: Situations may arise where a task "hogs" the processor, preventing tasks of the same priority from executing until the original task completes. To prevent this situation some p-System implementations permit the processor to rotate among tasks at the same priority level (assuming no higher priority task is ready to run).

NOTE: When a task is inserted into the ready queue, it is placed behind all other tasks having priorities greater than or equal to its own.

Priorities above a certain level are reserved for the operating system. This level is version-dependent; in Version IV.13 it is 254. The main task's priority (i.e., the priority of the program itself) is 128. Starting a process with a priority greater than 128 immediately suspends the main task. Conversely, starting a process having priority lower than 128 places the process on the ready queue but allows the main task to continue. In this case the started process will NEVER execute unless the program itself becomes suspended.

Starting a task with a priority of -1 causes the task to assume the "father" task's priority.

NOTE: The p-System utilizes a synchronous I/O system. Performing I/O generally does not suspend an executing task.

Examples of priority specification:

```
Program a;
Var PID: processid;
    I: integer;

  process p1;
  begin
    ...
  end;

begin
  I := 5;
  start(p1,PID,100);              { priority = 128 }
  start(p1,PID,100,-1);           { priority = 128 }
  start(p1,PID,100,90);           { priority = 90  }
  start(p1,PID,100,I*40);         { priority = 200 }
end.
```

3.0.2 Semaphores

Semaphores are variables declared with the predefined type SEMAPHORE. Semaphores are used solely for task synchronization; they are shared by tasks wishing to communicate with each other. Semaphores consist of two parts: a nonnegative integer counter and a queue for storing suspended tasks.

Generally a program associates a semaphore with a limited system resource, one that may or may not be available depending on how many other tasks are using it. The resource may be a physical device such as a terminal, which cannot be used by more than a limited number of tasks at once. It might be a data structure such as a collection of I/O buffers, which may be shared by a number of tasks but never by more tasks than there are available buffers. The semaphore's counter should be initialized to the maximum number of tasks allowed access to the resource at one time.

When a task accesses the resource it must decrement the semaphore counter, indicating that one fewer tasks are henceforth permitted access. When a task is finished using the resource it must increment the semaphore counter, indicating that the resource is henceforth available to one more task. The resource is being utilized to full capacity when the semaphore counter reaches zero. A task attempting to access a resource whose semaphore counter is zero is placed on the semaphore's wait queue along with any other tasks awaiting availability of that resource. Tasks on a wait queue are in a suspended state until the task using the resource relinquishes it, making it available to the task at the head of the wait queue.

NOTE: When a task is inserted into a semaphore's wait queue, it is placed behind all other tasks having task priorities greater than or equal to its own.

Semaphores are never accessed directly; they are accessed with semaphore operators. The principal semaphore operators are SEMINIT, WAIT, and SIGNAL.

SEMINIT initializes a semaphore variable by assigning it an initial count value and an empty wait queue.

WAIT checks the value of the semaphore count. If it is greater than zero, the count is decremented, and the current task (i.e., the one doing the WAIT) continues to execute. Otherwise, the current task is stopped; it is placed in the semaphore's wait queue, and becomes a suspended task. The task at the head of the ready queue then becomes the current task, and resumes its execution. Note that a task executing WAIT either continues as the current task or is stopped and becomes a suspended task.

SIGNAL examines the semaphore's wait queue. If it is empty, the semaphore's count is incremented. Otherwise, a suspended task is removed from the head of the wait queue and placed in the ready queue; it becomes a ready-to-run task. Note that when a task is moved from the wait queue to the ready queue it becomes the current task, replacing the task issuing the SIGNAL, if its priority is higher than the task issuing the SIGNAL. Thus, a task executing SIGNAL either continues as the current task or becomes a ready-to-run task.

Semaphores may be used in two ways, as binary semaphores and as counting semaphores. Binary semaphores have two states, as their counts only take on the values 0 and 1; they are used for mutual exclusion (section 3.0.2.1). Counting semaphores are so named because their count values can span the range of natural numbers; they are used for resource allocation (section 3.0.2.2). Binary and counting semaphores are both declared using the predefined type SEMAPHORE. The difference between them is in their context and usage.

WARNING: Semaphores must be initialized with SEMINIT before use; otherwise, system crashes may occur. Initializing a semaphore containing suspended tasks causes the suspended tasks to be lost by the system and incapable of further execution. A semaphore's count value must not exceed 32767; otherwise, the count value wraps around to a negative value, leaving the semaphore in an undefined state.

Sections 3.0.2.1 and 3.0.2.2 present standard uses of semaphores in concurrent systems. Section 3.0.2.1 describes **mutual exclusion**, which is used to protect global variables and routines from contention between tasks. Section 3.0.2.2 describes **task synchronization**, in which semaphores are used to synchronize the execution of a group of tasks.

3.0.2.1 Mutual Exclusion

When processes share a resource (usually a variable or an I/O device), it is often necessary to protect the resource from being accessed by more than one task at a time. This form of resource protection is known as **mutual exclusion**. Mutual exclusion is insured by placing all code which accesses the resource within a contiguous area of the program, called a "critical section".

Critical sections are implemented (using a binary semaphore) by preceding the critical code with a call to WAIT and terminating the critical code with a call to SIGNAL. The binary semaphore is initialized to 1, indicating that the critical section is initially open. When a task executes a critical section (by passing the WAIT), the semaphore count is guaranteed to be zero, ensuring that other tasks may not enter the critical section until it becomes available (when the executing task signals the semaphore).

Example of mutually exclusive use of a console screen:

```
Program example;
Var Console: semaphore;

  procedure ConWrite(OutIAm : integer; OutMsg: string);
  begin
    wait(Console); { start critical section }
                   { all access to console happens here }
    writeln('I am ', OutIAm,
                          ' and my message is ', OutMsg);
    signal(Console);   { end critical section }
  end;
```

```
    process MsgWriter(WhoIAm: integer; MyMsg: string);
    begin
      repeat
        ConWrite(WhoIAm, MyMsg)
      until false;
    end;

  begin
    seminit(Console,1);
    start(MsgWriter(1,'Shakespeare'));
    start(MsgWriter(2,'monkey'));
    start(MsgWriter(3,'typewriter'));
    .
    .
  end {example}.
```

This program contains three tasks, each of which executes the code of process `MsgWriter`. Each task has different values for the variables `WhoIAm` and `MyMsg`; each task attempts to continuously output the values of their variables to the console. Since only one task can use the console at a time, the writeln statement is isolated in procedure `ConWrite`, whose body is made a critical section. The tasks call on `ConWrite` whenever they need to read from or write to the console. The WAIT assures that the task currently executing `ConWrite` will not proceed with the writeln until a previous task completes its use of the console and executes the SIGNAL.

3.0.2.2 Synchronization

Semaphores may be used to synchronize the execution of a group of processes so that each process's execution depends on the actions of another process. Processes used in this fashion are known as **cooperating processes.** Cooperating processes are implemented by assigning a **private semaphore** to each process. A process considers its own private semaphore to represent an event which must occur before it can resume execution; therefore, the process waits on its private "event" semaphore. Processes wishing to indicate the occurrence of an event do so by signalling the corresponding private semaphore, thus activating the suspended process which owns the private semaphore.

Cooperating processes and private semaphores are illustrated in the example below, which demonstrates buffered data transmission (concurrency speeds up this activity by allowing simultaneous filling and sending of different data buffers). The resources in need of management are the `N` data buffers shared by the processes `FillBufs` and `SendBufs`. `FillBufs` finds an empty buffer and fills it with data. `SendBufs` finds a full buffer and dispatches it. The private semaphores are `BufAvail` and

`BufFull`. `BufAvail` indicates to `FillBufs` that a buffer has been sent and is available for filling; its initial value indicates that all buffers are initially available for filling. `BufFull` indicates to `SendBufs` that a buffer is full and available for transmission; its initial value reflects the lack of full buffers at the outset. The `FillBufs` task and the `SendBufs` task can each operate at their own individual pace; the semaphores assure that no attempt will be made to fill a buffer when none is available or to empty a buffer when none are full.

The variable `die` is used to terminate the two processes, and thus the program. `Die` is initially set to FALSE. `FillBufs` begins execution. As `FillBufs` fills buffers, `SendBufs` can begin to empty them. Presumably, some event will take place as either `FillBufs` or `SendBufs` executes to trigger the end of the filling/sending sequence. When the main task regains control of the processor it will detect that event in the IF statement and set `die` to TRUE. `FillBufs` and `Sendbufs` will each execute one more time as they regain control of the processor. Each task will then terminate upon encountering the `until die = true` clause and the program will conclude.

```
program Buffers;
const N = { number of available buffers };
var   BufFull, BufAvail: semaphore;
      die: boolean;

  process FillBufs;
  begin
    repeat
      wait(BufAvail);
      { ... Select an empty buffer and fill it ... }
      signal(BufFull);
    until die;
  end;

  process SendBufs;
  begin
    repeat
      wait(BufFull);
      { ... Select a full buffer and send it ... }
      signal(BufAvail);
    until die;
  end;

begin
  die := false;
  seminit(BufFull,0);
  seminit(BufAvail,N);
  start(FillBufs);
  start(SendBufs);
  if {and when we tire of playing with buffers}
    then die := true;
end {Buffers}.
```

3.0.3 Interrupts

The UCSD intrinsic ATTACH allows processes to be used as interrupt-driven device drivers. ATTACH assigns a logical (not physical) interrupt vector, or "event", to a semaphore; from then on, the semaphore is automatically signalled whenever the system receives an interrupt through the indicated interrupt vector.

It is important to avoid confusing events with hardware- generated interrupts. A hardware-generated interrupt is handled by the processor, typically by transferring control to an assembler code routine whose address is located at a hard-wired location. Since hardware- generated interrupts vary from machine to machine, they are not easily accessible from within the p-System in a machine-independent fashion.

In contrast, events are machine independent. Version IV of the p-System defines 64 events, numbered from 0 through 63, which can be attached to a semaphore from within a program. Events 0 through 31 are reserved for system use; the rest are for user use (but check your system documentation, since some implementations use a number of these values as well).

The association of an event with a "real" occurrence – as reflected by a hardware generated interrupt – happens within the p-code interpreter. Most p-System's are supplied with at least some of the events implemented. However if a user has a special device which must be associated with an unused event number, part of the interpreter I/O system will generally have to be rewritten. Consult your Installation Guide. Here are some predefined (but possibly unimplemented) Version IV system events:

```
0 - 16: Asynchronous I/O events (reserved)
    17: Soft break key
    18: Execution error occurred
    19: Keyboard character available
    20: Nil pointer reference occurred
    21: Clock
```

Version IV of the p-System allows for de-attaching a semaphore from an event by attaching it to another event or by attaching it to NIL. Other implementations do not provide a method for de-attaching semaphores; the system must be rebooted after running a user program containing attached semaphores if the devices causing interrupts cannot be disabled.

Version IV.13 of the p-System automatically de-attaches all events from program semaphores at program termination. It re-attaches events used by the operating system to the operating system's semaphores.

WARNING: Attaching an event to a semaphore causes that event to be de-attached from any semaphore it may previously have been attached to. It is impossible to re-attach the event to the original semaphore. Thus, in pre-IV.13 releases, if a user program attaches to an event the operating system had been attached to, it will be impossible to restore the event to the operating system. A system crash is the likely result. The Version IV p-System currently uses event 19 if the print spooler is enabled.

WARNING: Attaching to event 19 when the print spooler is enabled preempts the operating system's ability to read characters using the READ, READLN and GET intrinsics. Programs that attach to event 19 must read keyboard characters with the UNITREAD intrinsic described in section 4.44.

WARNING: ATTACH treats semaphore arguments as permanent variables; therefore, semaphores attached to interrupt vectors should be declared in the outer block of either the main program or the appropriate device process. The processor knows only of the memory address of an interrupt vector's attached semaphore, and continues to signal this address after every interrupt. It has no way of determining whether it is actually signalling a semaphore variable or merely damaging some unsuspecting code or data which happens to reside in memory previously occupied by an attached semaphore variable. Indiscreet use of ATTACH may adversely affect the system.

WARNING: Semaphores that have not been SEMINIT'ed should never be attached to an event; system crashes may occur.

NOTE: Simulation of concurrent processing exists only to the extent that interrupt-driven task switching occurs. Since task switching occurs in response to interrupts, the more frequently an interrupt occurs the better the simulation of concurrent tasks.

NOTE: Machines based on the Western Digital MicroEngine have a p-machine as their real processor. Thus, events are actually hardware interrupts and they do not adhere to the standard event numbers described above.

An example of interrupt processing may be found in section 6.10.

3.0.4 Time Slicing

Time slicing refers to the allocation of processor time to each task in the ready queue. As mentioned previously, tasks on the ready queue with the same priority as the current task will not be allocated processor time until the current task becomes suspended. However, certain implementations allow tasks of equal priority to rotate their turn at the processor. On these machines, time slicing is a side-effect of interactions between a system clock handler process and the task scheduling mechanism.

If a system clock is present it interrupts the processor 60 times per second. A clock handler process normally has higher priority than other tasks and continually waits for clock interrupts. When the processor receives a clock interrupt, a task switch occurs that activates the clock handler process, causing the current task to be inserted in the ready queue behind other tasks of equal priority. When the clock handler suspends itself, the processor selects the task at the head of the ready queue as the current task. Thus, the processor circulates between tasks of equal priority.

In the absence of a clock driver, task circulation is performed only as a result of a task blockage. A task blockage can occur at the execution of a WAIT intrinsic in the user program or as a result of certain system calls.

The following example demonstrates time slicing; when executed, the program prints final counts that are approximately equal, indicating that the tasks receive similar amounts of processor time.

```
program RaceCondition;
const limit = 1000;
      MyClockEvent = 21;  {may vary}
var   car1, car2, car3: integer;
      CheckeredFlag: boolean;
      ClockPID, PID: processid;

  process clock;
  var event: semaphore;
  begin
    seminit(event,0);
    attach(event,MyClockEvent);
    repeat
      wait(event);
    until CheckeredFlag;
  end;

  process racer(var counter: integer);
  begin
    counter := 0;
    repeat
      counter := counter + 1;
      if not CheckeredFlag then
        CheckeredFlag := counter >= limit;
      until CheckeredFlag;
      write(counter:6);
    end;

  begin
    CheckeredFlag := false;

    start(clock,ClockPID,500,250);

    start(racer(car1),PID,500);
    start(racer(car2),PID,500);
    start(racer(car3),PID,500);
  end.
```

This program contains three tasks executing the code of process `racer`. Note that each task has its own variable for keeping track of the number of times it executes the repeat loop. The global variable `CheckeredFlag` is set to true when one of the task's counters first reaches the limit (1000). This terminates the repeat loop in each task, and each task writes its final counter value.

The `clock` task is invoked whenever the clock generates an interrupt. Its high priority immediately grants it the processor, forcing a task switch. This causes the `racer` tasks to share the processor evenly.

NOTE: When a task is performing an I/O operation (e.g., waiting on input) all tasks are blocked and may not resume execution until the I/O operation completes. This "frozen" state may be avoided during input by polling the I/O device with the UNITSTATUS intrinsic to assure that the device is ready before initiating the I/O operation. See section 4.45 for UNITSTATUS details.

NOTE: Event processing may not occur during execution of assembly language routines or routines translated to native code by the Native Code Generator. Events signaled during this period are latched and then processed when p-code execution resumes.

3.1 Program Segmentation

Program segmentation refers to the division of program code by the programmer into disk-resident **code segments.** A code segment is memory-resident while it is executed; the system swaps it into and out of memory as necessary. Memory occupied by a code segment is freed for other uses when the segment is released, ensuring efficient use of memory; thus, segmented programs can avoid the memory constraints normally imposed on large programs.

Program segmentation in UCSD Pascal is achieved on a procedural basis through the use of **segments**. A procedure, function, or process is specified to reside in a separate code segment by preceding its declaration with the UCSD Pascal reserved word SEGMENT. The code segment contains the segment's code along with the code belonging to its (nonsegmented) local procedures. All code segments resulting from a single compilation reside in a single code file, which behaves identically to a similar file where segmentation was not employed. The only difference is that the routines of the segmented program might not be simultaneously resident in memory.

Although an in-depth discussion of the p-System memory management scheme is out of place here, a few general words regarding memory management and attendant version dependencies are in order.

Resident in low memory is an area called the heap, whose primary purpose is to contain system and program dynamic variables. The heap grows toward high memory.

The allocation of the rest of memory is highly version dependent. In pre-Version IV releases (including Apple Pascal), the rest of memory was occupied by a single entity called the stack. The stack began at the top of useable memory and grew downward. It contained two things: executable code segments and procedure activation records.

The first time a segment was called, it was loaded from disk onto the stack. Besides the p-code of the segment itself, the original procedure invocation also caused an activation record to have been built on the stack. This contained information linking the invocation of the segment procedure to the calling routine, as well as the procedure's local variables. As long as the segment procedure remained active, subsequent calls used the same copy of the code but each invocation caused a new activation record to be built on the stack. When the procedure returned to its caller the activation record was freed, but the p-code segment itself remained on the stack until there were no more active calls.

The stack and heap grew in opposite directions; when they met a (usually fatal) condition resulted, called "stack overflow".

Treatment of the stack changed radically with the release of Version IV of the p-System. In Version IV, the stack contains procedure activation records but not executable code. A new entity called the code pool contains all code segments. The code pool may occupy the space in memory between the stack and the heap, or exist in a different memory space, depending on available memory. When it becomes necessary to bring a segment into memory from disk, the system attempts to find room for it in the code pool. If there is no contiguous area in the code pool large enough to contain the segment, the system moves the other segments around (and possibly out) in an attempt to squeeze the new segment in. It may even remove segments that have active invocations. Previous implementations locked a segment into its original memory location for the duration of its residence.

The code pool construct provides a convenient means of supporting hardware that can address greater than 64k bytes of memory. The Version IV p-machine uses 16 bit addresses and thus can directly access no more than 64kb. However, since the code pool is now distinct from the stack and heap, the p-machine can access it in a different 64kb "bank", effectively providing support for 128kb – half for the stack and heap and half for the code pool. A code pool contained inside the stack/heap space is called an internal code pool; it competes with the stack and heap for memory. Code pools located in alternate banks are called external code pools. Note that presently the implementation details for extended memory are highly hardware dependent.

The Version IV implementation affords better memory utilization at a cost of speed. At unpredictable times, a program may slow down as the system attempts to find memory space for a segment. However, this overhead is reduced somewhat because the system maintains activity statistics and uses them to keep the most frequently called segments in memory. Memory is better utilized with this scheme because the system need only maintain the currently executing segment in memory. It may cache inactive segments depending on available memory.

The MEMLOCK and MEMSWAP intrinsics were introduced with Version IV to give the programmer a measure of control over when segments are removed from memory. See the next section for details.

Programs should be segmented with an eye towards minimizing the frequency with which segments must be loaded from disk. A good strategy is to keep the most frequently executed code in memory, and to provide the system with the greatest flexibility in choosing which segments to discard, if there is no room for all. A good candidate for segmentation is initialization code, which is usually executed only once at the beginning of a program.

Segments must be independent of each other in order to reap the benefits of segmentation. For example, envision a large piece of code requiring division into two segments (named `A` and `B`) in order to conserve memory. Version IV requires that both the calling and called segments be in memory at the time of the call. If the logic of the program is such that the only caller of `B` is `A`, segmentation is fruitless-both code segments must be memory-resident while `B` is called. `A` proper division results in mutually independent segments which are called sequentially. Program segmentation is most effective when it influences the design of large programs (as opposed to "tuning" existing programs).

NOTE: Within the main program or any segment declaration, the code comprising local segments must appear before code belonging to the enclosing segment or program. As can be seen in the example, this does not prevent unsegmented procedures from containing local segments, but does affect the order in which local procedures are declared. As with unsegmented procedures, segments may be declared forward to resolve interprocedural references. Forward declaration of segmented and unsegmented procedures may occur in any order.

NOTE: A Version IV p-System program may contain between 1 and 255 code segments. Version II programs are limited to 7 segments, although various schemes have been developed to extend this limitation. Other implementations may have other upper limits; consult your documentation. See also section 3.2 on the unit construct.

WARNING: In general, the maximum size of a segment is 64kb. However, certain Version IV implementations have further restrictions on the maximum size of a segment. In particular, current PDP/LSI-11 implementations with an external code pool restrict the size of a segment to 16kb. The compiler does not assure that the maximum size of a segment is not exceeded. Attempts to execute a code file which uses an overly large segment may result in spurious "Segment Not Found" messages.

WARNING: When a program calls a disk-resident segment, the disk volume containing the program's code file (and thus its code segments) must be online and mounted in the same drive as when the program was started; otherwise, an error message may appear or the system may crash.

Example of segment declarations:

```
program main;

  procedure p1; forward;
  segment function p2: integer; forward;

|-segment procedure p3;
|
| |-procedure p3p1;
| |
| | |-segment function p3p1p1:boolean;
| | | begin
| | |   ...
| | |-end {p3p1p1};
| |
| | begin
| |   ...
| |-end {p3p1};
```

```
|
| begin
|   ...
|-end {p3};

|-segment function p2 {: integer};
| begin
|   p1;  {This use of p1 necessitates
|         its forward declaration}
|   ...
|-end;

|-procedure p1;  {not a segment procedure;
|                 must appear after p3 and p2}
| begin
|   ...
|-end;

begin
  ...
end {main}.
```

This example program contains four segments: `main`, `p2`, `p3` and `p3p1p1`. Note that the code in a main program always occupies its own segment, though this is not explicitly declared in the program. A segment procedure may be nested to a maximum depth of 7, and will behave as a "normal" procedure except for its memory residence characteristics, as noted previously. Procedure `p1` is not a segment procedure; it is part of the segment `main`. It must therefore appear after segment procedures `p2` and `p3`. However, it is used by `p2` and must therefore be declared forward.

In this example `p2` happens also to be declared forward. Note the syntax – the word "segment" must appear both in the forward declaration and in the beginning of the procedure itself.

3.1.1 Alternate Segment Management Strategies

A segment is normally guaranteed memory-residence only while it is executing. The UCSD intrinsics MEMLOCK and MEMSWAP provide for more sophisticated segment management strategies. These intrinsics allow runtime control over the loading and unloading of segments.

MEMLOCK accepts a string value parameter containing a list of segments to be "locked" into memory until explicitly "unlocked". MEMSWAP accepts a similar string parameter containing a list of segments to be "unlocked", and removed from the code pool when necessary. The segment list may contain segment and unit identifiers (section 3.2) declared in the program and its used units, or in the operating system. Identifiers are separated by commas; spaces and invalid identifiers in the segment list are ignored. The form for a MEMLOCK or a MEMSWAP call is:

```
<memlock-call> ::= MEMLOCK(<segment-list>)
<memswap-call> ::= MEMSWAP(<segment-list>)

<segment-list> ::= <segment-name>
                   {,<segment-name>} | <empty>
```

The MEMLOCK intrinsic causes each code segment in the segment list to be read into memory and locked into the system code pool. Subsequent calls to such segments use the MEMLOCKed copy of the code rather than loading it from disk. MEMLOCKing an already MEMLOCKed segment MEMLOCKs it further. Matching calls to MEMSWAP are necessary to render the segment swappable.

NOTE: All MEMLOCKed user segments are made swappable at program termination.

WARNING: Attempts to MEMLOCK a segment whose identifier is shared by more than one segment has unpredictable results. Segments with identical names may come into being as a result of separate compilations. But they may also come into being in the same compilation, as follows:

```
program foon;
  segment procedure twit;
    segment procedure greep;
    begin
    end;
  begin {twit}
  end;

  segment procedure greep;
  begin
  end;
begin
end.
```

Pascal syntax is not violated since the first occurrence of segment procedure `greep` is nested within `twit` while the second is global.

WARNING: Indiscreet use of MEMLOCK may render the heap incapable of containing large buffers on a system containing an internal code pool.

Example of MEMLOCK and MEMSWAP use:

```
program mems;

  segment procedure seg1;
```

```
    begin
      <... segment code ...>
    end;

    segment procedure seg2;
    begin
      <... segment code ...>
      memswap('seg1');
              {Will release seg1 when seg2 is invoked}
    end;

  begin
    memlock('seg1, seg2');  {Both segments are loaded}
    seg2;                   {Seg1 is now released...}
    memswap('seg2');        {... and so is seg2}
  end {main}.
```

See section 5.0.11 for alternate strategies for Apple Pascal.

3.1.2 Segments and Tasks

In Version III of the p-System a few restrictions are imposed on the use of segments in conjunction with concurrent tasks. These restrictions are due to architectural limitations of that version.

Processes may be declared as segments; however, they operate somewhat differently. If the process code executed by a task is declared as a segment, the code segment containing the process is read onto the task's stack, and remains there until the task terminates execution. Unfortunately, when the main program terminates, the system is unable to shut down segmented tasks in an orderly fashion, and so must be rebooted; therefore, in Version III segment processes should only be used in dedicated (i.e., nonterminating) programs.

3.2 Separate Compilation

Separate compilation (also known as "external compilation" or "modular programming") allows programs to be created from individually compiled modules. Some advantages resulting from separate compilation are:

- New modules can be written, compiled, and combined with existing modules to create new programs. The new modules themselves might later be used in other programs. Thus, a growing library of precompiled software tools may become available for use in general software development.

- Large programs constructed from separate modules are easily modified; changes are isolated to individual modules, allowing fast and reliable program maintenance. Programs may be designed by teams of programmers, with each member of the team able to code, compile and test his module separately.
- Programs can be developed that are larger than could otherwise be compiled in one piece on the system.

Separately compiled modules are built in UCSD Pascal using the UNIT construct. Unit declaration is described in section 3.2.1. Section 3.2.2 explains how units are referenced by host programs. Section 3.2.3 provides information on unit linkage.

NOTE: This section provides a program-level description of units. Section 6.12 describes the philosophy and pragmatics of unit construction and usage.

3.2.1 Units

Units are collections of uses-, constant-, type-, variable-, procedure-, function-, and process- declarations grouped to address a specific class of related problems. These objects may be referenced by either a program or other units. Programs and units which use other units are called **hosts.** Units consist of four parts: an **interface section**, an **implementation section**, an **initialization section** and a **termination section**. Objects declared in a unit's interface section are public; they are accessible to both the unit and the host which uses the unit. Objects declared in the implementation section are private. They are accessible only within the unit. The initialization section is a code sequence that usually initializes unit variables and is automatically executed once at program invocation time. The termination section is a code sequence that is automatically executed once at program termination time and usually performs any "shut-down" operations required. It is executed after the termination code of the host which uses the unit.

An example of a unit declaration appears on the next page. Note that the interface section may contain procedure and function headings, but routine bodies are not allowed. Procedure and function headings in the interface section are similar to forward declarations; when the corresponding routines are defined in the implementation section, the parameter list is omitted.

```
unit mnemones;
interface
```

```
  type mnemone = (truth, beauty, wisdom, knowledge, etc);

  procedure relapse;
  { forget all items learned }

  procedure learn (newentry: mnemone);
  { learn a new item }

  function recall (look: mnemone): boolean;
  { has item been learned? }

implementation

  type entryptr = ↑listentry;
       listentry = record
                     data: mnemone;
                     next: entryptr;
                   end;
   var listhead: entryptr;

   procedure relapse;
   begin listhead := nil end;

   procedure learn;
   var entry: ↑listentry;
   begin
     new (entry);
     with entry↑ do
       begin data := newentry; next := listhead; end;
     listhead := entry;
   end;

   function recall;
   var entry: ↑listentry;
   begin
     recall := false; entry := listhead;
     while entry <> nil do
       if entry↑.data <> look then
         entry := entry↑.next
       else recall := true;
   end;

begin
  listhead := nil;      {initialization section}
***;
  relapse;              {termination section}
end {mnemones}.
```

A major purpose of a unit is to allow a program to perform high-level operations on abstract information. The program is not necessarily aware of the data structures used to represent the information. It knows which operations are defined on the information, but does not need to know how those operations were implemented.

Unit `mnemones` allows a program to emulate (in a tongue-in-cheek fashion) a person's struggle to acquire knowledge. Areas of wisdom which may be dealt with are given in type `mnemone` in the interface section; a program using this unit may declare entities of this type.

The areas of wisdom may be learned individually and forgotten collectively; the program may also seek to remember whether or not an item has been learned (and not forgotten). The procedures `learn` and `relapse`, and the function `recall`, are declared in the interface section and are therefore available for a program to use.

It is apparent from the implementation section that a linked list is used to represent the items learned, but a program using this unit does not have to be aware of this. In fact, the record which defines elements of the list (type `listentry`) is not accessible to a program using `mnemones`, nor can the program directly access the linked list itself; the variable `listhead`, as well as all variables local to the routines in the implementation section, may be manipulated only by those routines.

When execution of a program using `mnemones` begins, the variable `listhead` is initialized to NIL in the unit's initialization section. Before the program concludes all the heap space allocated by the unit is deallocated in the unit's termination section. (Deallocation of heap variables immediately prior to program termination is redundant because the p-System automatically RELEASEs all heap variables after program termination. Nevertheless, it is a good programming practice).

NOTE: The method of deallocation used in the `relapse` procedure is wasteful in that it does not provide for the reallocation of deallocated memory. See section 3.5 for further details.

The syntax for unit definition is shown below (it is loosely based on the Pascal syntax in Appendix D of the *User Manual and Report*).

```
<compilation unit>    ::= <program> | <library>

<program>             ::= <program heading>;
                          <inline unit part>
                          <uses part>
                          <block>.

<library>             ::= <unit definition>
                          {;<unit definition>}.

<inline unit part>    ::= {<unit definition>;}

<uses part>           ::= [USES <unit id-part>
                          {,<unit id-part>}

<unit id-part>        ::= <unit identifier>[<identifier>
                          {,<identifier>}]
```

```
<unit identifier>      ::= <identifier>

<unit definition>      ::= UNIT <unit identifier>;
                           <interface part>
     (Required in IV.1)    [<implementation part>]
     (IV and Apple)        [BEGIN
     (IV and Apple)         [<initialization section>]
     (IV only)              [***;
     (IV only)               [<termination section>]]]
     (IV and Apple)        END

<interface part>       ::= INTERFACE
                           <declarations>
                           <procedure/function headings>

<implementation part>  ::= IMPLEMENTATION
                           <declarations>
                           <procedure and function bodies>

<declarations>         ::= <uses part>
                           <constant definition part>
                           <type definition part>
                           <variable declaration part>
```

The ***; statement should be used only to separate initialization and termination sections. It should not be contained in any statement or procedure body. The compiler does not enforce this rule, but violating it can lead to bizarre results.

Labels may not be declared globally in units, nor may GOTO statements occur in either the unit initialization section or termination section; if they do occur, the results are unpredictable. EXIT(PROGRAM) is the only legal means to arbitrarily transfer control in the initialization or termination sections of a unit. In the initialization section EXIT(PROGRAM) causes control to pass directly to the termination section of that unit. In the termination section it causes control to pass to the next termination section, if any. If there are no pending termination sections the program is ended. EXIT with a unit identifier is not permitted. Segment declarations are allowed in the implementation section only; they follow the conventions described in section 2.4 for forward declarations and procedure body declarations.

NOTE: The compiler may emit initialization and termination code even if these sections are not declared by the programmer. The presence of file declarations in either the interface or implementation sections causes the generation of hidden code that initializes and closes these variables. Additionally, the use of the EXIT intrinsic anywhere in the implementation section causes the generation of hidden initialization code.

NOTE: The presence of initialization and termination sections directly influence the amount of time required to initiate and terminate the execution of a host program. Units containing such code must be read into memory and executed before and after the host program executes. The proliferation of initialization and termination code can therefore lead to significant delays in program startup and shutdown.

NOTE: The ***; statement is not available in pre-Version IV releases. The initialization section BEGIN is available only in Version IV and Apple releases. Version IV.1 *requires* an implementation section.

NOTE: The INTERFACE text of a unit is stored in the unit's code file and therefore occupies disk space. After a host using the unit is compiled, however, the INTERFACE text is no longer required. The Library utility may be used to remove the INTERFACE text from production code files. Consult your system documentation for details.

A unit may consist solely of an interface section (and possibly an initialization and/or termination section); this is known as a **data unit.** A data unit consists of only uses, constant, type and variable declarations which are accessible to a host program. The responsibility of providing operations on the data is left to the host program or another unit. Example of a data unit:

```
unit ComplexData;
interface
  type complex = record
                   realpart, imaginary : real;
                 end;
  var one, i : complex;
begin   {initialization section}
  one.realpart := 1;  one.imaginary := 0;
  i.realpart := 0;    i.imaginary := 1;
end. {ComplexData}
```

WARNING: Intuitively, the initialization code of any unit used by a host should be executed before the initialization code of any host. In most versions of UCSD Pascal this order is not guaranteed and should not be relied upon. Hence, unit initialization code should not rely on the values of variables initialized by the initialization code of other units.

The compiler accepts the following combinations of units and programs during a single compilation:

- A program.

- A unit (as in the previous example).
- A group of units.
- A program containing one or more inline units.

NOTE: If a file containing separate programs and/or units is submitted to the compiler, only the first program or unit in the file will be compiled.

NOTE: An interface section may be contained in an include file if the keyword INTERFACE is also contained in the include file. However, interface sections may not contain include file directives.

3.2.2 Using Units

A unit may be used in a host by naming it in a USES statement. In programs, the USES statement must appear after the program heading. In units, the USES statement must appear at the beginning of either the interface section or the implementation section. Objects declared in the interface section of a used unit become globally declared objects within the host. Objects imported by using a unit in the implementation section remain private to the host unit.

WARNING: Although the compiler issues no error message if a program USES a unit of the same name, executing the compiled program will cause an immediate stack overflow! For example:

```
program DontDoThis;
uses DontDoThis;
begin
end.
```

Version IV.1 of the p-System introduced a feature called selective USES. This feature permits the host to specify the identifiers to be imported from a used unit. At compile time, symbol table space is allocated to represent only the specified identifiers and their component types. Since this usually represents a subset of the identifiers provided in the unit's interface section, a compile-time space savings results.

Selective USES is employed by following the USES <unit identifier> with a list of identifiers enclosed in parentheses. The identifiers may be any constant, type, variable, procedure, function or process present in the unit's interface section. Only those identifiers named will be exported from the unit; the programmer must assure that identifiers from which the selected identifiers are derived are also selected (e.g., If one type definition is derived from another type or constant, the parent type or constant must be included if the derived type is to be included).

Code segments for units are normally resident in memory only when needed (similar to segment procedures), unless operated upon by the MEMLOCK intrinsic (see section 3.1).

NOTE: In situations where a used unit uses other units in its interface section, the program must name the nested units in its USES statement before naming the unit which uses them. For example, if unit `A` uses unit `B` in its interface section, then a host program using unit `A` must contain the USES statement `uses B,A;`.

WARNING: Because identifiers imported from used units have global scope within the host, naming conflicts may arise between globally declared program identifiers and identifiers imported from used units. A convention commonly used to avoid conflict is to prefix all identifiers exported from a unit with a unique character sequence (e.g., `SC_Home`, where the prefix `SC` indicates that the procedure came from the screen control unit).

NOTE: One copy of a unit's public and private variables exists for all USES of a unit by a program and its used units. This presents a potential problem when, for example, unit `A` uses units `B` and `C` and unit `B` itself also uses unit `C`. The variables of unit `C` may undergo manipulation by both unit `A` and unit `B`– with neither `A` nor `B` aware of the other's interference.

In the following example, the program uses the `mnemones` unit declared in a previous example; identifiers imported from the unit are underlined for emphasis. Note that the initialization section of the unit is executed before the program is executed.

```
program UnitDemo;
uses mnemones;
type charset = set of char;
var  finished: boolean;

  function GetCommand(valid: charset): char;
  var ch: char
  begin
    repeat
      read(keyboard,ch);
    until ch in valid;
    GetCommand := ch;
    writeln(ch);
  end;
  function GetCategory(command: string): mnemone;
  begin
    write(command,
          ': T(ruth B(eauty W(isdom K(nowledge E(tc');
    case GetCommand(['T','B','W','K','E']) of
      'T': GetCategory := truth;
      'B': GetCategory := beauty;
      'W': GetCategory := wisdom;
      'K': GetCategory := knowledge;
      'E': GetCategory := etc;
    end;
  end;

begin {UnitDemo}
  finished := false;
  repeat
    write(
      'Education: L(earn R(ecall F(orget G(raduate');
    case GetCommand(['L','R','F','G']) of
      'L': learn(GetCategory('Learn'));
      'R': if recall(GetCategory('Recall')) then
             writeln('Remembered')
           else
             writeln('Forgotten');
      'F': relapse;
      'G': finished := true;
    end;
  until finished;
end.
```

This program exemplifies not only the use of units, but also typical UCSD Pascal programming style. The body of the program is little more than a case statement, selecting one of the options L(earn, R(ecall, F(orget or G(raduate. Responsibility for presenting a prompt and returning with a mnemone value always belongs to the single procedure `GetCategory`. Validation of user input is always handled by the procedure `GetCommand`.

3.2.3 Unit Linkage

Linkage to units is performed both at compile time and at runtime. At compile time, the compiler imports the identifiers contained in a used unit's interface section. At runtime, the operating system loads code segments of used units and resolves unit references. Linkage information is maintained with unit code, which is found either in the code file resulting from compilation of the unit or in a library containing the code of the unit. A library is merely a number of code files which have been combined into a single file using the system library utility. The p-Systemallows a default library called SYSTEM.LIBRARY which may contain whichever units the user wishes to place there. The user may also define any number of additional library files; see the discussion of USERLIB.TEXT, below.

In order to import the interface section of a unit used by a host, the compiler must first locate the unit. The host may directly specify the name of a file containing the unit's code with the $U compiler option (see section 5.0.3). This file is searched first for the units used. If a unit is not there, or if $U is not specified, the compiler searches the current code file (for in-line units), then SYSTEM.LIBRARY on the system volume. If the search is unsuccessful, a compiler syntax error is emitted.

If a unit is modified, it should be recompiled and reinstalled into the library search path. If only the implementation section is changed, hosts using the unit need not be recompiled. If a unit's interface section is modified, all hosts using the unit must be recompiled with the new version.

WARNING: Executing a program that USES units whose INTERFACE sections have been modified since the program's compilation can produce unexpected or fatal results. This occurs because the positions of unit variables and procedures relative to the beginning of the unit are determined during the compilation of a host that uses that unit. Executing a host that calls a unit whose variable or procedures have been reorganized in the INTERFACE section causes the host to access incorrect variables or procedures. Note that because variable offsets and procedure numbers are assigned in the order they are encountered by the compiler, adding variables and procedures to the end of the variable and procedure declarations (instead of the middle) does not invalidate references in the host, and therefore does not mandate recompilation of the host.

In order to execute a program using units, the operating system must locate each unit used by the program and its used units. This search is performed in the following manner: First, the code file itself, then the default library, then, in sequence, the files enumerated in USERLIB.TEXT. If a unit is not found, the system emits an error message and aborts the execution of the program.

In Version IV of the p-System, the user libraries are specified, in a text file called USERLIB.TEXT. This file contains a list of all the files the system should search in the quest for used units, in the desired search order.

NOTE: The Version IV redirection feature permits the runtime re-specification of the name of the user library text file. Therefore, any text file can replace USERLIB.TEXT as the list of files to be searched for used units. Consult the Version IV Users Manual for further details.

NOTE: Units are compiled into code segments in a manner similar to programs; the unit occupies one segment and each segment procedure or function occupies a segment. Programs and units may directly access up to 255 units and segments.

In Version II and Apple Pascal, UNITs were implemented somewhat differently from the Version IV implementation. The ***; statement and the termination sections were not available. In Version II, the initialization sections were not available either.

In Version II and Apple Pascal UNITs were compilable separately from their hosts, but had to be explicitly linked to the host using the system Linker utility.

Apple Pascal releases include a facility called the INTRINSIC UNIT. INTRINSIC UNITs are placed in the system library using the Library utility and do not require explicit linking; they are linked to the host dynamically at runtime in a manner similar to the Version IV implementation. Each INTRINSIC UNIT is treated as a segment, however there is a limitation of sixteen intrinsic units per system.

Further, each such INTRINSIC UNIT has a specific segment number associated with it; it has to fit into that specific "slot" of the host program during execution. These numbers must be determined by the programmer when the INTRINSIC UNIT is compiled. This scheme causes difficulties when a host must use a number of INTRINSIC UNITs compiled with the same segment number.

3.3 Files

UCSD Pascal provides a number of extensions for file handling. The extensions include:

- Direct access to the file system from programs.
- Interactive file I/O on the system terminal.
- Random-access disk files.
- Block-oriented files for systems programming.

A number of extensions are also provided to allow access directly to a device, without regard to its structure. These are discussed in section 3.9.

Section 3.3.1 introduces the UCSD intrinsic CLOSE and describes extensions made to the standard procedures RESET and REWRITE. Together, these intrinsics allow programs to access the file system. Section 3.3.2 describes the predeclared file type INTERACTIVE; when applied to interactive files, the standard procedures READ and RESET are redefined to accommodate interactive I/O. Section 3.3.3 describes the predeclared file KEYBOARD, which reads characters from the standard input without echoing them to the standard output. Section 3.3.4 describes block files. Block files are accessed with the UCSD intrinsics BLOCKREAD and BLOCKWRITE which read and write data in integral numbers of blocks. Block files allow efficient manipulation of large, arbitrarily structured files. Section 3.3.5 introduces the UCSD intrinsic SEEK, which is used to randomly access the contents of disk files.

3.3.1 File System Access

UCSD Pascal provides direct access to the file system, allowing programs to manipulate disk files and perform file operations on I/O devices. It is useful to make a distinction between file variables and **external files** – a file variable is a logical entity declared in a program, while an external file is either a physical disk file or I/O device. File system access is accomplished by connecting a program's file variable with an external file. A file is **open** if it has been connected, and **closed** if it either has not yet been connected or has been connected and subsequently disconnected. File I/O operations may be performed only on open files.

The file system is accessed with the intrinsics RESET, REWRITE, and CLOSE. RESET and REWRITE connect files, while CLOSE disconnects files.

REWRITE creates new files. Its form is:

```
<rewrite-call> ::= REWRITE(<fileid>[,<filename>])
```

where <fileid> is a file variable identifier and <filename> is a string constant or variable containing a file name. REWRITE creates a new external file with the given file name and prepares the file variable for subsequent file operations.

NOTE: As mentioned in the file system specification, files on a disk volume must have distinct file names; an existing file is automatically deleted if another file with the same name is entered in the disk directory. A disk file created by REWRITE is assigned temporary status; it becomes a permanent file (and an old file with the same name on the same volume is deleted) only if it is closed and locked (see below for details). Thus, programs which generate temporary files need not worry about inadvertently deleting permanent disk files.

RESET opens existing files for subsequent file operations, and resets the file to its beginning position. RESET may be applied to already open files, in which case the file is reset to its beginning position. The form for

RESET is:

```
<reset-call> ::= RESET(<fileid>[,<filename>])
```

where <fileid> is a file variable identifier and <filename> is a string containing a file name. Calling RESET with the second parameter present opens an existing external file named by <filename> and prepares the file variable for subsequent file operations. If the file named by <filename> is not present a runtime error results (unless I/O checking is suppressed; see section 5.0.6 for details). Note that RESET with a single parameter (i.e., the file identifier) works as defined in Standard Pascal.

WARNING: As in Standard Pascal, performing a RESET causes an automatic call to GET. When the file identifier is connected to a serial device, the program hangs until the GET is satisfied by incoming data. See section 3.3.2 for more details.

NOTE: Applying the RESET intrinsic to a blocked file does not perform an implicit GET. See section 3.3.4 for details.

CLOSE disconnects files. The form for CLOSE is:

```
<close-call> ::= CLOSE(<fileid>[,<option>])

<option> ::= NORMAL | LOCK | PURGE | CRUNCH
```

The options determine the final state of a file. NORMAL (which is the default option) preserves pre-existing files which were RESET, but deletes files newly created by REWRITE. LOCK preserves files as permanent disk files. Locking a newly created file may delete an existing permanent file if they share the same name and reside on the same volume. PURGE deletes the file associated with <fileid> from the directory. Both temporary files (opened with REWRITE) and permanent files (opened with RESET) are deleted when the PURGE option is utilized. CRUNCH is equivalent to LOCK, but causes the file to be truncated at its current position.

NOTE: An implicit CLOSE(<file>, NORMAL) is performed on files which are not explicitly closed before the procedure in which they are declared terminates.

NOTE: The UCSD intrinsics OPENOLD and OPENNEW are synonymous with RESET and REWRITE respectively, and were used in pre-Version IV releases of UCSD Pascal.

Chapter 4 contains detailed descriptions of the intrinsics mentioned in this section. Chapter 7 describes the file system and file naming conventions.

Examples of file system access using RESET, REWRITE, and CLOSE:

```
program FileDemo;
var infile,outfile: text;
    st: string;  {Limits line size to 80 chars}
begin

  { open up the disk file named "master.text" }
  reset(infile,'master.text');

  { copy to a disk file named "copy1.text" }
  rewrite(outfile,'copy1.text');
  while not eof(infile) do begin
    readln(infile,st);
    writeln(outfile,st);
  end;
  close(outfile,lock);

  { rewind master file for second pass }
  reset(infile);

  { copy to a disk file named "copy2.text" }
  rewrite(outfile,'copy2.text');
  while not eof(infile) do begin
    readln(infile,st);
    writeln(outfile,st);
  end;
  close(outfile,lock);

  { close down master file }
  close(infile,normal);
end.
```

`Infile` and `outfile` are logical files of type text. This is a predeclared type which is equivalent to file of char. RESET is used to associate `infile` with the existing file `master.text`. REWRITE associates `outfile` first with `copy1.text`, then with `copy2.text`. Since the LOCK option is used, each of these files is added to the volume directory upon execution of the respective CLOSE statement. (Files may be copied more efficiently using the BLOCKREAD and BLOCKWRITE intrinsics, discussed in section 3.3.4.)

Note that files with a suffix of .TEXT have a special significance in the p-System; they possess a header record with information necessary in order to work with the file using any of the p-System editors. This is transparent, however, when doing normal READ or WRITE operations from or to the file.

3.3.2 Interactive Files

UCSD Pascal provides the predeclared file type INTERACTIVE in order to facilitate the use of the system terminal as an input file. Interactive files are structurally equivalent to text files; the only difference between them is the manner in which the standard procedures RESET, READ, and READLN are defined to act.

To explain the need for interactive files, it is first necessary to examine the definitions of text file operations in Standard Pascal. Let `ch` be a character variable, and `f` a file of type TEXT; the following rules then hold for RESET and READ:

- RESET(f) is defined to perform an implicit GET(f)
- READ(f,ch) is equivalent to ch := f^; GET(f)

where GET advances the file pointer `f↑` to the next character in the file.

Using these standard definitions, the following program attempts to create a simple console prompt by writing a prompt message to the console screen and accepting a response from the console keyboard:

```
program prompter;
var infile,outfile: text;
    ans: char;
begin
  reset(infile,'console:');
  rewrite(outfile,'console:');
  write(outfile,'Are you sure this will work (y/n) ?');
  read(infile,ans);
  if answer = 'y'
     then writeln ('yes')
     else writeln ('no ');
end.
```

Unfortunately, this program doesn't work as expected; RESET performs an implicit GET, so the program will appear to "hang" until a character is typed on the console. *After* a character is typed, the prompt appears; however, READ will assign the contents of the file buffer to the variable `ans` before executing another GET. Thus it will use the character previously typed to satisfy the RESET operation. It will *then* pause to GET another character into the buffer variable, but this character will be ignored since the program contains no further READ. A 'yes' or 'no ' will appear on the screen following input of the second character, but it will reflect the value of the first character! The program is obviously ill-suited for interactive use.

With an interactive file i, the following rules hold for RESET and READ:

- RESET(i) does not perform an implicit GET(i)
- READ(i,ch) is equivalent to GET(i); ch := i^

The program shown above executes more reasonably if `infile` is declared with type INTERACTIVE. The program does not hang when the input file is opened, and the prompt response is not read until after the prompt message is displayed. Only one character needs be input; it will be assigned to the variable `ans`, and will be the basis for the 'yes' or 'no ' that appears on the screen.

The definition of interactive files affects the manner in which the standard functions EOLN and EOF are used. The following code fragments are functionally equivalent (`f` is a file of type text, `i` is a file of type interactive, and `ch` is a character variable).

```
while not eoln(f) do           while not eoln(i) do
  read(f,ch);                    read(i,ch);
read(f,ch); {EOLN marker}
```

Both loops read characters from the file until the end of a line is reached, as signaled by the EOLN marker in the window variable. But with file `f`, the window variable `f↑` will always contain the *next* character for `ch`. Thus, an additional read is necessary to flush the end-of-line marker. With file `i`, `i↑` and `ch` contain the same character after each READ; thus, `ch` contains the end-of-line marker when EOLN(i) evaluates to true.

3.3.3 The Keyboard File

UCSD Pascal contains the predeclared file KEYBOARD for reading characters directly from the terminal keyboard. KEYBOARD is an interactive file, and is the non-echoing equivalent to the predeclared file INPUT. For example, given `ch` as a character variable, the statements:

```
read(keyboard,ch);
write(output,ch);
```

are equivalent to `read(input,ch)`, assuming no redirection of output. The KEYBOARD file is useful when it is desired to verify a character before echoing it, or when it is necessary to echo a character other than the one typed.

NOTE: EOF(KEYBOARD) becomes true only after typing <null>. The console end-of-file command is read as a normal character.

NOTE: The KEYBOARD file and INPUT file are distinct files with distinct states. While a READ from INPUT determines the EOLN and EOF function values, it does not affect the EOLN(KEYBOARD) and EOF(KEYBOARD) values, and vice-versa.

3.3.4 Block Files

Block files allow low-level access to the file system; they are intended for system programming. Block files are declared with the predeclared type FILE and may be accessed only with the BLOCKREAD and BLOCKWRITE intrinsics. These are integer-valued functions. They accept as parameters a block file identifier, a buffer, a number of blocks and (optionally) a starting block number, and return the number of blocks actually transferred. (A block is 512 bytes long.) The optional starting block number parameter allows disk files to be randomly accessed by block number; in its absence, successive block I/O operations access consecutive blocks. A disk file is viewed as a group of contiguous blocks; the first block is block 0.

NOTE: The RESET intrinsic does not perform an implicit GET when applied to block files.

WARNING: No range checking is performed on the size of the buffer variable. Therefore, it is possible to read data beyond the end of a buffer by specifying a greater block count than the buffer can hold.

Example of block I/O using explicit I/O checks and implicit starting block:

NOTE: The function IORESULT is discussed in section 4.16. It returns an integer indicating the completion status of an input or output operation. The system normally aborts a program with a runtime error when an I/O operation completes abnormally. The $I compile option, discussed in section 5.0.6, allows the program to continue after an abnormal I/O completion; the program may then examine the value of IORESULT to determine the appropriate response.

```
program FileCopy1;
const blksexpected = 1;
var  infile,outfile: file;
     buf: packed array [1..512] of char;
                       {no type checking with block I/O; }
     junk, blksread: integer;
                       {file may contain data of any type}
     endofile: boolean;
begin
```

```
  endofile := false;
  reset(infile,'source.data');
  rewrite(outfile,'dest.data');
  while not endofile do
    begin
      {$I-  Turn off system I/O checking}
      blksread := blockread(infile,buf,blksexpected);
      if ioresult <> 0 then writeln('disk read error');
      endofile := blksread <> blksexpected;
      if not endofile then
        begin
          junk := blockwrite(outfile,buf,1);
          if ioresult <> 0 then writeln('disk write error');
        end;
      {$I↑  Restore system I/O checking to previous state}
    end;
  close(infile);
  close(outfile,lock);
end.
```

Example of block I/O using implicit I/O checks and explicit starting block:

```
program FileCopy2;
const N = 5;
                          {process N blocks at a time}
var   infile,outfile: file;
      buf: packed array [1..N,1..512] of char;
                          {provides N x 512 bytes}
      blknum, blksread: integer;
begin
  reset(infile,'source.data');
  rewrite(outfile,'dest.data');
  blknum := 0;
  repeat
    blksread := blockread(infile,buf,N,blknum);
    if blockwrite(outfile,buf,blksread,blknum) <> 0
      then;
    blknum := blknum + N;
  until blksread < N;
  close(infile);
  close(outfile,lock);
end.
```

3.3.5 Random Access Files

UCSD Pascal provides the SEEK intrinsic for random record access in a structured disk file. SEEK accepts two parameters: a file identifier, and an integer indicating the record to be accessed. SEEK moves the file window so that a subsequent GET or PUT operation accesses the specified record. The first record in a file is record 0.

NOTE: In Standard Pascal, an open file is either read or written exclusively. Random access files in UCSD Pascal are opened with REWRITE for new files and RESET for pre-existing files, but in either case they can be both read (using GET) and written (using PUT).

NOTE: The standard procedure EOF can be used to check if the specified record number exceeds the number of records in the file. Calling SEEK itself always sets EOF to false, but a subsequent GET operation reveals the presence or absence of a record in the file window. If GET causes EOF to become true, the file window is past the end of the file, and the buffer variable is undefined.

WARNING: SEEK disregards the end of a file when setting a new file position. After seeking to a record position past the end of a file, PUT should be called only if the file window immediately follows the last record in the file; otherwise, the file state becomes undefined and adjacent files may be destroyed. It is wise for a program to explicitly track the current last record in a randomly accessed file.

NOTE: UCSD Pascal disk files occupy integral 512-byte blocks. In the case of structured files the operating system automatically maintains (in the volume directory) the ending position of the last logical record in the last block of a file. However, this information is not maintained for files created using UNIT or BLOCK I/O. Since the SEEK intrinsic uses this information, SEEK cannot be applied to UNITWRITE-created files or BLOCKWRITE-created files.

Example of SEEK:

```
program DataBase;
var f: file of string;
    recnum: integer;
begin
  reset(f,'string.data');
  repeat
    write('Enter record number (-1 terminates) : ');
    readln(recnum);
    if recnum < 0 then exit(program);
    seek(f,recnum);
    get(f);
    if eof(f) then writeln(' No such record')
    else
      begin
        writeln(' Current value is: ',f↑);
        seek(f,recnum);   { reseek record for update }
        write(' Enter new value: ');
        readln(f↑);
```

```
          put(f);
        end;
    until false;
    close(f,lock);
  end {DataBase};
```

This program allows on-line updating of selected records in the file `string.data`. Note that no checking need be done on the value entered for record number; SEEK is immediately invoked with that value, followed by GET. If the value supplied is greater than the largest current record number, GET will set `eof(f)` to true and the program will not attempt to display or modify the (non-existent) record. If a valid record number is supplied, the record is displayed. Before the record can be updated, SEEK must again be invoked with the same record number since the previous GET has advanced the file window to the record beyond the one to be updated.

3.4 Strings

UCSD Pascal contains the predeclared data type STRING. Variables and constants of type STRING contain character sequences. The length of a character sequence stored in a string variable may vary during the execution of a program. A number of operations are provided for strings:

- The file operators READ, READLN, WRITE and WRITELN accept string arguments.
- The intrinsics CONCAT, COPY, DELETE, INSERT, LENGTH and POS (discussed in chapter 4) perform common string operations.
- Individual characters in a string variable may be accessed similarly to an array of characters.
- All comparison operators (e.g., <>) accept string arguments.

Although the length of the character sequence stored in a string variable may vary during the execution of a program, the actual amount of memory occupied is static. String types are declared with a static length attribute. The default static length is 80 characters. Static length attributes are explicitly assigned by following the predeclared identifier STRING with an unsigned integer constant (denoting the static length) enclosed in square brackets ([ ]). The maximum length attribute is 255 characters.

The dynamic length of a string is stored in an "extra" byte preceding the rest of the string. This byte is normally transparent to the programmer.

The dynamic length of a string may not exceed its static length. An attempt to assign a large string of characters to a smaller string variable will result in an execution error (i.e., " String Overflow "). There is no automatic truncation in UCSD Pascal.

Examples of string type declarations:

```
type  normal = string;             {default static length}

      volname = string[7];         {static length = 7 chars}

      bigstring = string[255];     {static length = 255 chars}
```

NOTE: Static length attributes allow users to minimize the amount of space allocated to strings (disk space with respect to files containing strings; memory space with respect to string variables). Strings are type-compatible regardless of their static length attribute.

NOTE: String variables are always allocated an even number of bytes, including a hidden length byte. Thus, `STRING[10]` and `STRING[11]` both occupy 12 bytes.

Example of string assignment:

```
s1 := 'this is a string constant';
s2 := s1;
```

NOTE: String constants may not exceed 80 characters.

Individual characters within a string may be referenced by indexing into the string variable (e.g., `s1[5]`– note that string variables are equivalent to a PACKED ARRAY OF CHAR in this respect). Valid string indices range from 1 to the current dynamic length of the string; indices outside of this range cause an execution error (i.e., "Invalid Index") to occur.

Example of an invalid string index:

```
s1 := '1234';
s1[5] := '5';
```

NOTE: If the length of a string is 0 (i.e., its value is ''), any string indexing causes an execution error.

NOTE: In p-System Version IV.1 and later releases the $R compiler option will suppress range error checks on string subscripts. Suppression of this error was not possible in pre-Version IV.1 releases.

NOTE: The dynamic length of a string may be set explicitly by disabling index range checking and storing the desired length into the length byte:

```
var
  s: string[80];
  NewLen: integer;
begin
  NewLen := 12; {or whatever}
  {$R-}
  s[0] := chr(NewLen);
  {chr function must be used to
   convert length to a single byte}
  {$R↑}
```

The relational operators =, <>, <, <=, >, >= yield a boolean result when applied to string operands. Comparisons are performed lexicographically (e.g., word order in a dictionary). Note that trailing spaces are significant. Appendix I displays the character order.

Examples of string comparison:

```
if 'write' < 'writeln'
  then writeln('strings compares work');

if s1 = s2 then writeln('string vars equal');

if 'language' = 'language  '  {note trailing blank}
   then writeln('This would be PL/I')
   else writeln('But this is UCSD Pascal!');
```

When a value for a string variable is read using READ or READLN, all characters up to an end-of-line character (carriage return) are assigned to the string variable. When a string is entered from the console it must be terminated by a carriage return, whether or not the input was performed with READ or READLN. By definition, READLN swallows the carriage return. READ, however, leaves the carriage return as the window variable; it is picked up by the next read operation. As a result, it is suggested that strings be read from the console only with READLN. Additionally, attempting to read two strings with one READLN call will fail because the end-of-line that terminates the first string input is not flushed until the end of the READLN. Hence the second string will receive an empty input.

Character strings longer than the static length of a string variable are truncated by READLN before being assigned to the string variable.

UCSD Pascal provides the following intrinsics for string manipulation: CONCAT, COPY, DELETE, INSERT, LENGTH and POS. CONCAT accepts two or more strings as arguments and returns a single string containing the concatenation of the string arguments. COPY extracts a character sequence (a "substring") from a string and returns the sequence as a string. INSERT stuffs a string value into another string. DELETE removes characters from a string. LENGTH returns an integer containing the dynamic length of a string. POS returns an integer denoting the starting position of a character pattern within a string.

Example of string intrinsics:

```
program strings;
var s1, s2, s3: string;
    int: integer;
begin
  s1 := 'The quick brown system';
  s2 := 'jumped over the lazy document';

  writeln(length(s1),' ',length('Q'));

  int := pos('brown',s1);
  writeln(int,' ',pos(s1,s2));

  s3 := concat(s1,' ',s2);
  writeln(s3);

  writeln(copy(s1,1,4),copy(s2,pos('document',s2),8));

  writeln(s1);
  s3 := 'quick brown';
  delete(s1,pos(s3,s1),length(s3));
  writeln(s1);

  insert(' is a moving target',s1,succ(length(s1)));
  writeln(s1);
end {strings}.

** Program output **

22 1
11 0
The quick brown system jumped over the lazy document
The document
The quick brown system
The  system
The  system is a moving target
```

3.4.1 String Parameters

Strings may be passed as value or variable parameters; however, the compatibility of strings having different static lengths can cause some subtle problems.

First, note that string types possessing a length attribute specification are considered structured types, and thus may not appear in the formal parameter list of a procedure or function; according to Standard Pascal, only type identifiers may appear here.

Example of strings as formal parameters:

```
type bigstring = string[132];

procedure trans(param1: string; param2: bigstring);
```

Strings passed as value parameters (rather than as VAR parameters) are copied into local data areas by the called procedure. The code for this task is produced automatically by the compiler; it is executed when the procedure is first entered. If the actual parameter's dynamic length exceeds the formal parameter's static length, the execution error "String too long" occurs when the string copy is attempted.

Example of an execution error during string copying:

```
program example;
type shortstring = string[4];

  procedure crash(param: shortstring);
  begin
    { string-copying code causes error here }
  end;

begin
  crash('oversized actual parameter');
end {example}.
```

WARNING: Strings passed as variable (VAR) parameters can cause serious problems as a result of a lapse in UCSD Pascal type-checking. Formal parameter references within a procedure become indirect references to the actual string parameter. Within the procedure, however, the formal parameter's static length attribute overrides the actual string's static length. If the formal parameter's dynamic length exceeds the actual string's static length, the formal parameter may be assigned values that overrun the string's data space without causing an execution error. This results in either a system crash or damage to the contents of an adjacent variable.

Example of integrity violation from poor type checking:

```
program features;

type bigstring = string[250];

var smallstring: string[10];
    victim: string;
      {will be corrupted because it follows smallstring}

  procedure whackstring(var param: bigstring);
```

```
    begin
      param := 'this string is larger than ten characters';
    end;       {          ↑characters from here
                           on will overwrite victim!}

  begin
    victim := 'this string will be overwritten';
    writeln('before: ',victim);
    whackstring(smallstring);
    writeln('after: ',victim);
  end {features}.
```

In the program above, the string `victim` immediately follows the string `smallstring` in memory. `Smallstring` has a static length of 10 characters. If more than 10 characters are entered into `smallstring` the excess characters will extend beyond `smallstring`, overwriting `victim`.

Normally, runtime range checking prevents more than 10 characters from being assigned to `smallstring`. In the illustration above, however, `smallstring` is passed as a parameter to procedure `whackstring`, where the formal parameter corresponding to it (`param`) has a static length of 250 characters. Since `param` is a VAR parameter, any modifications made to it within procedure `whackstring` actually affect `smallstring` itself. However, the runtime system only enforces the static length of param – 250 characters. It does not enforce the static length of the actual parameter, `smallstring`. Thus, the assignment of a value longer than 10 characters to `param` causes no runtime error. But it does cause memory locations following `smallstring` to become corrupted. `Victim` will be overwritten by the excess characters of the string assigned to `param`.

3.5 Dynamic Variable Management

UCSD Pascal provides two sets of intrinsics for dynamic variable allocation and deallocation: the UCSD Version II intrinsics and the additional intrinsics provided by UCSD Version IV. The Version II intrinsics have the advantage that they are common to all versions of UCSD Pascal. However, the Version IV intrinsics allow the deallocation of single dynamic variables and provide support for variable-sized buffer allocation. The Version II intrinsics are NEW, MARK and RELEASE (section 3.5.1). The intrinsics introduced in Version IV include VARNEW, DISPOSE, VARDISPOSE and VARAVAIL (section 3.5.2).

3.5.1 The Version II Heap

All dynamic variable allocation is performed in an area of memory known as the **heap.** The heap starts in low memory and grows towards high memory (where low memory begins at location 0 and high memory > low memory). The system stack, which contains procedure code, local variables and parameters, starts in high memory and grows towards low memory. The NEW intrinsic is used for the allocation of a single dynamically allocated variable. Successive calls to NEW allocate variables in successive ascending memory locations, thus advancing the heap towards the stack. If the heap and stack collide, a stack overflow error occurs.

As in Standard Pascal, NEW accepts any number of arguments. The first argument is required; it is a pointer to the type of variable being allocated, and is returned containing the address of the allocated item. The remainder of the arguments are optional. They are used when the variable being allocated is a variant record with variants of different sizes. NEW normally allocates as much space as is necessary to contain the largest variant. However, if NEW is given the tag field names for nested variants as the second through last arguments it will allocate only the amount of space necessary to contain the variant specified.

WARNING: The p-System does not perform a runtime check to assure that assignments to variant records utilize valid variants. Thus, a carelessly written program may supply a value to a dynamically allocated variant record that is larger than the amount of space allocated to the record. A system crash or corruption of surrounding variables may result.

```
program ZapMemory;
type
  itemp = ↑item;
  item = record
         case i: integer of
           1: (smallvariant: integer);
           2: (largevariant: string);
         end;
var
  killer: itemp;
begin
  new(killer,1);  {corresponds to smallvariant}
  killer↑.largevariant := 'Goodbye, Mr. Bits';
end.
```

In program `ZapMemory`, TYPE `item` has two variants, an integer (2 bytes long) and a string (82 bytes long). The NEW allocates a record of a size to accommodate the smaller variant. The assignment statement, however, assigns a value much larger than 2 bytes to the record! Thus, surrounding heap locations will be corrupted.

The MARK and RELEASE intrinsics are used for the deallocation of dynamically allocated variables. MARK and RELEASE accept pointer variables of any type as arguments. Given a pointer variable `p`, `MARK(p)` opens a new heap for dynamically allocated variables. The heap is identified by the value assigned to `p` by MARK. Subsequent calls to NEW allocate dynamic variables only in the new heap. `RELEASE(p)` deallocates all dynamic variables in the heap designated by `p`.

Essentially, MARK causes the system to note a heap location by saving the current value of its internal top-of-heap pointer in the variable `p`. The top-of-heap pointer advances as NEW is used to allocate additional dynamic variables, but RELEASE causes the system to set its internal top-of-heap pointer back to the value saved in `p`, freeing all variables allocated since execution of the corresponding MARK.

NOTE: New heaps are allocated within the current heap; thus, heaps are nested. Deallocating a given heap results in the deallocation of all subsequently opened heaps.

WARNING: Careless use of MARK and RELEASE can lead to "dangling references" (i.e., pointer variables pointing to deallocated dynamic variables). Use of dangling references can cause unpredictable results, including system crashes.

NOTE: In pre-Version IV releases, MARK and RELEASE do not check the validity of their arguments. The programmer must assure that pointers passed to MARK are used for no purpose other than an argument to a subsequent call to RELEASE. Pointers passed to RELEASE must be initialized by a previous call to MARK.

Example of MARK and RELEASE:

```
program dynamic;
type citizenptr = ↑citizen;
     citizen = record
                  name: string;
                  number: integer;
                  neighbor: citizenptr;
               end;
var mrklist, listhead: citizenptr;

  procedure add(cloname: string; ID: integer);
  var cloneunit: citizenptr;
  begin
    new(cloneunit);
    with cloneunit↑ do
      begin
        name := cloname;
        number := ID;
        neighbor := listhead;
      end;
    listhead := cloneunit;
  end {add};

begin
  mark(mrklist);           {allocate space for list}
  listhead := nil;
  add('Clone, Norman Q.      ',2763);
  add('Dumptruck, T.         ',2764);
  add('Maton, Otto F. S.     ',2765);
  add('Kaboozee, Kitzle N.   ',2766);
  listhead := nil;
  release(mrklist);        {deallocate entire list}
end {dynamic}.
```

This program builds, then destroys, a linked list of records of type `citizen`. The pointer variable `listhead` is used to maintain the current beginning of the list, and changes as records are added to the list by procedure `add`. Before `add` is invoked in the main program, the top-of-heap is recorded in the pointer variable `list`, via a call to MARK. A number of records are added to the list. Prior to program termination, a call to RELEASE with pointer `list` restores the top-of-heap to its original value, thus destroying the linked list.

Although the pointer `mrklist` is declared as type `citizenptr` in this illustration, it could just as well have been declared as a pointer to any type (e.g., `↑integer`). This is because `mrklist` is not used to point to data but serves merely to mark a location on the heap.

Note that the variable `listhead` is explicitly set to NIL, even though the entire linked list it points to has been RELEASEd. RELEASE merely sets back the top-of-heap pointer. It does not change values on the heap, nor does it modify pointers to the RELEASEd heap locations. In fact, were `listhead` not set to NIL, the program would still be able to access the linked list. Of course, any additional variables allocated on the heap would overwrite the linked list. Programmers should assure that they do not attempt to illegally use the heap in this manner. However, to avoid situations leading to such misuse a program may explicitly set to NIL any pointers to RELEASEd heap locations.

NOTE: Releasing the heap at the end of a program is unnecessary since the operating system also deallocates all dynamic variables when a program completes execution.

3.5.2 The Version IV Heap

The Version IV heap implementation provides all Version II heap intrinsics. It also includes the DISPOSE, VARNEW, VARDISPOSE and VARAVAIL intrinsics.

Dissimilarities between the Version II.0 heap and the Version IV heap occur as a result of the DISPOSE intrinsic. This intrinsic is used for the deallocation of a single dynamically allocated variable. The memory space occupied by the deallocated variable is recycled by subsequent calls to NEW, assuming the space occurs in the current heap and it is large enough to accommodate the new variable. Note that successive calls to NEW are not guaranteed to allocate variables adjacently in memory, as is the case with the Version II.0 heap.

The VARNEW and VARDISPOSE intrinsics are used for the allocation and deallocation of variable-sized buffers. They accept two parameters: a pointer variable of any type and an unsigned integer representing the number of words to be allocated. (Unsigned integers are discussed in section 6.2.) The VARNEW function attempts to allocate a buffer of the requested number of words. If there is sufficient memory for such a buffer, the pointer is returned pointing to the buffer and the requested word count is returned as the function value; otherwise the function value is zero. The VARDISPOSE procedure deallocates a buffer of the specified size at the specified pointer location.

The VARAVAIL function accepts a string value containing a list of segments and returns the size of the largest available memory space, assuming all specified segments are memory-resident. The segment list is of the same form as that used by the MEMLOCK intrinsic described in section 3.1.

NOTE: Pointer variables passed to the RELEASE, DISPOSE and VARDISPOSE intrinsics are returned containing the value NIL.

NOTE: Pointer values passed to RELEASE must be the result of a prior MARK; otherwise, an "Invalid Heap Operation" error occurs.

NOTE: Calls to DISPOSE or VARDISPOSE must be made with the same size structure as was used with the corresponding NEW or VARNEW call; otherwise, a system crash may occur. In the event a variant of a record was allocated, the program is responsible for assuring that the same variant is DISPOSEd.

NOTE: Calling NEW or VARNEW to allocate a one-word structure actually allocates two words; corresponding calls to DISPOSE and VARDISPOSE deallocate two words. Calls to MARK allocate six words in addition to a new heap; calls to RELEASE deallocate that space.

Example of extended memory management usage:

```
program extended;
type buffer =
  record
    case integer of
      2: (twoblocks  : array [0..511] of integer);
      4: (fourblocks : array [0..1023] of integer);
    end;

  procedure method1(size: integer);
  var bufptr: ↑buffer;
  begin
    if size = 2 then
      new (bufptr, 2)
    else
      new (bufptr, 4);
    {use buffer for something}
    if size = 2 then
      dispose (bufptr, 2)
    else
      dispose (bufptr, 4);
  end;

  procedure method2(size: integer);
  var bufptr: ↑buffer;
  begin
    if varnew (bufptr, size*256) <> 0 then
      begin
        {use buffer for something}
        vardispose (bufptr, size*256);
      end;
  end;

begin
  method1(2);
  method2(4);
end {extended}.
```

Two methods are illustrated in this program for allocating and deallocating a variable sized buffer. Procedure `method1` utilizes the standard Pascal NEW and DISPOSE procedures. The size in blocks of the desired buffer is passed as a parameter to `method1`. `Method1` then makes use of the fact that TYPE `buffer` was deliberately declared as a variant record with integer tags 1 and 2. `If` size has value 2 then the variant with tag 2 of `buffer` is allocated. This variant is two blocks long. If `size` has value 4 then the four block variant of `buffer` is allocated.

The problem with the approach of `method1` is that if there are more than two desired variations in buffer size, the additional variations would require corresponding variants in type buffer. Procedure `method2`, however, uses the UCSD intrinsics VARNEW and VARDISPOSE to allocate and deallocate the buffer. These permit exact specification of the number of words desired; this value is independent of the size of any variant. Thus, the desired size in blocks passed to `method2` is multiplied by 256 to yield the size in words, and that number of words is VARNEWd directly. Procedure `method2` does not require that TYPE `buffer` be a variant record. It will nevertheless permit allocation of a buffer of *any* size.

See sections 3.11.9 and 6.4 for further discussion of variable buffer allocation.

3.6 Extended Precision Arithmetic

UCSD Pascal provides a data type known as the "long integer" for extended precision arithmetic. Long integers are used like standard integers, but may contain values greater than maxint.

Long integer types are defined by appending a length attribute to the predeclared type INTEGER. Length attributes are similar to those used in string types: an unsigned integer constant delimited by square brackets ([and]). The maximum length attribute is 36.

Length attributes are used to minimize the amount of space allocated to long integers (disk space with respect to files containing long integers; memory space with respect to long integer variables). Long integers are type compatible regardless of their length attribute.

The interpretation of long integer length attributes is hardware dependent. On many machines, the length attribute is taken to mean the maximum number of digits expected. Implementations for other processors interpret the length attribute as a specification of the number of bytes to be used. This often implies that values with considerably more than that number of decimal digits can be represented. Thus, the length attribute should be regarded as the least possible number of digits which may be contained in the long integer.

Examples of long integer type definitions:

```
type  shortint = integer[3];
      longint  = integer[36];      { max size }
```

Depending on their value, constants defined as integers become either integer or long integer constants. Constants in the range -32767..32767 default to integer constants; constants outside this range are treated as long integer constants.

Under pre-Version IV UCSD Pascal systems, programs with long integer constants compiled on one processor may not run properly on a different processor. Recompilation may be required.

Caution should be exercised when creating data files using long integers which may be ported to a different machine. The internal representation of a long integer varies from machine to machine.

Examples of integer constants:

```
const Rydberg = 10973731;    { long integer }
      Hoffman = 0;           { integer      }
```

In general, long integers may be used anywhere it is syntactically correct to use REAL types. For instance, long integers and integers may be mixed in arithmetic expressions; integers are implicitly converted to long integers in mixed expressions. Integers may be assigned to long integers; however, long integers must be explicitly converted to integers (with the standard function TRUNC, described below.) Note that direct conversion between long integers and reals is not provided.

WARNING: See Appendix G for some problems arising from the use of mixed expressions.

The arithmetic operators +, -, * and DIV yield a long integer result when applied to long integer operands. (Note that MOD is not defined.) The relational operators =, <>, <, <=, > and >= yield a boolean result when applied to long integer operands.

Unlike integers, long integers enforce overflow checking; when a long integer variable is assigned a value larger than it can contain, the execution error "Integer Overflow" occurs.

WARNING: Intermediate expression results should not exceed the maximum permitted number of long integer digits (36) – integer overflow may not be detected.

Example of a program using long integers:

```
program example;
{generates LOTS of powers of two}
var long: integer[36];
begin
  long := 1;
  repeat
    writeln(long);
    long := long * 2;
  until long > 20000000;
end.
```

All file I/O operators (including READ and WRITE) accept long integer arguments.

WARNING: On some pre-Version IV UCSD Pascal systems backspace does not work when reading a long integer from the console.

The standard function TRUNC is extended to accept long integers as arguments (as with reals, an execution error occurs if the argument is outside the appropriate range). The UCSD intrinsic STR converts long integers to strings. Given a long integer `L` and a string `S`, `STR(L,S)` assigns to `S` a character string representation of the value in `L` (complete with minus sign, if required).

Example of STR:

```
program money;
type bucks = integer[30];
var CashFlow: bucks;

  procedure PrintDough(amount: bucks);
  var dollars: string;
  begin
    str(amount, dollars);
    insert('.', dollars, pred(length(dollars)));
    writeln('$',dollars);
  end {PrintDough};

begin
  CashFlow := 2323972233;
  PrintDough(CashFlow);
  PrintDough(199);
end {money}.

**Program output**

$23239722.33
$1.99
```

Long integers are often used to represent large monetary values. These values may be stored and manipulated as integral numbers of pennies. They are converted to dollars-and-cents values by inserting a decimal point prior to printing. This program demonstrates a simple procedure to accomplish the insertion. `PrintDough` accepts a long integer parameter, and converts it to a string using the STR intrinsic. The INSERT intrinsic is then used to edit in a decimal point in front of the second digit from the end of the string.

3.6.1 Long Integer Parameters

Long integers may be passed as value and variable parameters; however, the compatibility of types with different length attributes may cause some subtle problems.

First, note that long integer types are considered structured types, and thus may not appear in the formal parameter list of a procedure or function; according to standard Pascal, only type identifiers may appear here.

Example of long integers as formal parameters:

```
type longint = integer[32];
procedure trans(param1, param2: longint);
```

When long integers are passed as variable parameters, long integer types with different length attributes lose their type compatibility. The formal and actual parameter types must possess identical length attributes.

Long integers passed as value parameters are adjusted by the calling routine to the size declared in the called routine's formal parameter list. The code for this task is produced automatically by the compiler; it is executed by the calling procedure. If the value of the actual parameter is too large to fit in the formal parameter, the execution error "Integer overflow" occurs when the long integer is adjusted.

Example of an execution error during parameter adjust:

```
program example;
type shortint = integer[4];

  procedure crash(param: shortint);
  begin
    { ... }
  end;

begin
  crash(32948759384 75);
    { adjust code causes error here;        }
    { this value is too large for integer[4] }
end {example}.
```

3.7 Extended Comparisons

UCSD Pascal extends the relational operators to accept pointer, array and record types as operands.

3.7.1 Records and Arrays

The relational operators = and <> yield a boolean result when applied to array and record operands. Operands must be type compatible (see section 2.14). Operators compare entire structured variables. Structures are equal if and only if the fields comprising the structures are equal.

WARNING: Structured comparisons are implemented on a word-to-word basis, from the beginning of the structure until the end. Field boundaries are not considered. Therefore, it is useless to compare structured variables which fail to completely utilize their allocated data space. Relational operators should not be used in the following cases:

- Records containing string types.
- Many packed arrays and records.

Data Space for strings is allocated statically, and string values expand and contract in their fixed data area at runtime. The area between the end of a string value and the end of its data space is undefined, but is considered significant in a structured comparison. Thus, comparison of records containing strings does not work correctly.

The UCSD Pascal compiler's packing algorithm may leave unused bit fields in the words comprising the data space allocated for packed records and arrays. Because the unused bit fields contain undefined values, comparison of packed records and arrays may not work correctly. The exceptions to this restriction are byte arrays (e.g., PACKED ARRAY OF CHAR) and packed variables which (by chance or design) completely utilize their allocated data space. See section 6.0 for a description of the packing algorithm.

Note also that two structures which may appear identical in structure may not be. This is because fields may or not be allocated in the order in which they are declared. See section 6.0 for details.

Example of record and array comparison:

```
program compare;
var a,b: record
           i,j: integer;
           r: real;
         end;
    p,q: record
           str: string[5];
           r: real;
         end;
    x,y: array[0..150] of integer;
    count: integer;
begin
  for count := 0 to 150 do
    begin x[count] := 4; y[count] := 4 end;
  if (x = y) then writeln('x and y ARE the same ')
             else writeln('x and y are different');

  with a do begin i := 4; j := 6; r := 3.14159 end;
  with b do begin i := 4; j := 6; r := 2.71828 end;
  if (a = b) then writeln('a and b are the same ')
             else writeln('a and b ARE different');
    with p do begin str := 'not5';  r := 3.14159 end;
    with q do begin str := 'not5';  r := 3.14159 end;
    if (p = q) then writeln('p and q are the same   ')
               else writeln('p and q ARE?! different');

  end {compare}.
```

The program will report that x and y *are* the same, and that a and b *are* different, as they are. These structures do not contain strings, and there is no "empty" space between fields. However, the program will also report that p and q *are* different (?), even though the fields within them were set to identical values. The reason is that the field str is allocated space for five characters, but only four are assigned. The fifth character was never initialized, and would (very likely) contain different values in p and q. Nevertheless, these bytes would be considered in the comparison and p and q would be reported as different.

3.7.2 Pointers

The relational operators =, <>, <, <=, > and >= yield a boolean result when applied to pointer operands. These operators are implemented as unsigned integer comparisons.

3.8 Byte Array Manipulation

UCSD Pascal provides the intrinsics MOVELEFT, MOVERIGHT, SCAN, FILLCHAR and SIZEOF for efficient manipulation of large arrays of data. MOVELEFT and MOVERIGHT perform mass movement of data within arrays. FILLCHAR initializes arrays. SCAN searches an array for the presence (or absence of) a byte value. These intrinsics are intended for use with byte (e.g., character) arrays; however, the lack of type checking on their parameters allows them to be used as general purpose data manipulators (with the understanding that the price of freedom is responsibility). These intrinsics are byte-oriented: address parameters are resolved to byte addresses and parameters specify byte counts (see detailed explanation of parameters further on in this section). Many of the intrinsics have parameters which require byte values; these are characters, or integers in the range 0..255.

WARNING: Count values are treated as signed integers. Negative count values in MOVELEFT, MOVERIGHT and FILLCHAR calls are treated as as zero byte counts, not as unsigned integers. Operations on structures larger than 32767 bytes must be split into two parts.

SIZEOF is a (compile-time) function which accepts either a variable or type identifier as an argument and returns an integer value indicating the number of bytes allocated for the data type denoted by the identifier. SIZEOF shifts the burden of determining the size of a data type onto the compiler, thus making it safer and easier to use the byte array intrinsics.

A compile-time function has its effect only during compilation. It does not generate code. The compiler evaluates the SIZEOF function by referencing the compiler symbol table and replaces SIZEOF with the appropriate integer value. Thus, SIZEOF may be used freely in a program without fear of incurring runtime overhead.

SIZEOF may be used only on entire arrays, not on individual array elements. Further, the compiler does not permit SIZEOF to be applied to structures within structures unless a WITH is used. See the following program:

```
program measure;
type a = record
           z: integer;
           b: record
                x: string;
                y: real;
              end;
        end;
     d = packed array[0..1] of char;
     e = integer;
var  aa: a;
     dd: d;
begin
    writeln(sizeof(a));  { <— these three all compile}
    writeln(sizeof(aa));
    writeln(sizeof(e));

 {  writeln(sizeof(aa.b)); <— this one does not compile ..}
    with aa do writeln(sizeof(b)); { <— .. while this one
                                                 compiles! }

 {  writeln(sizeof(d[0]));          <— does not compile}
end.
```

NOTE: If a record contains variant fields, SIZEOF uses the largest variant when determining the size of the record.

WARNING: SIZEOF ignores pointer de-references. For example:

```
SIZEOF(p↑)
```

returns the size of the pointer variable `p` rather than the size of the object to which `p` points; in this case, it is necessary to pass SIZEOF the identifier denoting `p`'s base type.

FILLCHAR accepts a starting address, integer byte count and byte value. Beginning with the starting address, it initializes <byte count> bytes to the indicated byte value. Regardless of the type of the variable whose starting address is supplied, FILLCHAR treats the destination as an array of bytes.

MOVELEFT and MOVERIGHT perform mass movement of data; both accept a source address, destination address and integer byte count. The source and destination addresses are normally array elements. The bytes between the source address and the address calculated by <source address> + <byte count> - 1 comprise the source array. The bytes between the destination address and the address calculated by <dest address> + <byte count> - 1 comprise the destination array.

MOVELEFT and MOVERIGHT move data from the source array to the destination array one byte at a time. MOVELEFT starts at the *lower-addressed* end of both arrays and copies bytes traveling toward higher addresses. It is recommended for use when the source and destination arrays overlap, and the source array is lower in memory than the destination array. By starting at the lower-addressed locations, MOVELEFT prevents the first bytes moved from overwriting other source bytes – those in the overlapping area. Such an overwrite would be disastrous since the source bytes in the overlapping area haven't yet been moved to their destinations. MOVERIGHT starts at the *higher-addressed* end of both arrays and copies bytes traveling toward lower addresses. It is recommended for use when the source and destination arrays overlap, and the source array is higher in memory than the destination array. By starting at the higher-addressed locations, MOVERIGHT prevents the first bytes moved from overwriting other source bytes – those in the overlapping area. Such an overwrite would be disastrous since the source bytes in the overlapping area haven't yet been moved to their destinations.

NOTE: Movement of data blocks between nonoverlapping arrays is usually performed with MOVELEFT, as it represents a more natural style of moving data. Certain combinations of MOVELEFT and MOVERIGHT with overlapping source and destination addresses produce complex results; their use is not recommended without some forethought (see example below).

WARNING: Array indices are treated as signed integers. Use of an index whose value is less than the declared lower bound of the the source or destination array may yield unexpected or fatal results.

Example of byte array manipulators:

```
program blockmove;
var source1, source2: packed array[0..511] of char;
    dest: packed array[0..1023] of char;
    int: integer;
begin
  fillchar(source1, sizeof(source1), 0);
                       {fill source1 bytes with 0000 0000}
  fillchar(source2, sizeof(source2), 1);
                       {fill source2 bytes with 0000 0001}
  moveleft(source1[0], dest[0], 512);
                       {fill 1st half of dest with source1}
  moveright(source2[0], dest[512], 512);
                       {fill 2nd half of dest with source2}
  moveleft(dest[512], int, 2);
                       {get 1st two occurrences of 0000 0001
                        of dest, into int                 }
  writeln(int);
                       {value of int is 0000 0001 0000 0001
                        so 257 will be printed            }
end.
```

No overlapping of source and destination arrays occurs in this example. The array `dest` is filled with `source1` and `source2`, and `int` is filled with two bytes from `dest`. Note that no type checking occurs in the movement of the two bytes from a packed array of char to an integer. However, it is the programmer's responsibility to assure that no more than two bytes are moved, since an integer is two bytes long.

Example of shady use of MOVELEFT and MOVERIGHT:

```
program boggle;
var bytes: packed array [1..30] of char;
begin
  bytes := 'this is the text in this array';
  writeln ('123456789012345678901234567890');
  writeln(bytes);
  moveright(bytes[10],bytes[1],10);
  writeln(bytes);
  moveleft(bytes[1],bytes[3],10);
  writeln(bytes);
  moveleft(bytes[23],bytes[2],8);
  writeln(bytes);
end.

** Program Output **

123456789012345678901234567890
this is the text in this array
he text ine text in this array
hehehehehehetext in this array
his arrayehetext in this array
```

It is left as an exercise for the reader to trace through execution of this program!

WARNING: When using FILLCHAR, MOVELEFT or MOVERIGHT with dynamically allocated buffers, be sure to specify the buffer, *not* the pointer to the buffer, as the parameter. Thus, the following fragment is incorrect:

```
var
  buffer: packed array[0..511] of char;
  b: ↑buffer;
begin
  fillchar(b,512,55);
```

The buffer will not be filled here. Instead, the 512 bytes beginning at the *pointer* to the buffer will be overwritten, with disastrous results. The correct use of FILLCHAR in this situation would be

```
fillchar(b↑,512,55);
```

SCAN is a function which accepts an integer scan length, a partial boolean expression and the starting address of what will be treated as an array of bytes. It is used to determine the first byte value in a byte array that matches (or, alternatively, does *not* match) a specified target value.

A partial boolean expression has the following form:

```
<partial boolean expression> ::= <relop> <target>
<relop> ::= "=" | "<>"
<target> ::= <character variable> | <character constant>
```

In plain English, partial boolean expressions are an = or <> followed by the target character variable or constant; they appear as half of a boolean expression. The missing left operand is defined to be each byte in the byte array in succession, beginning at the specified starting address. SCAN examines each byte until either it finds a byte value satisfying the partial expression or the scan length is exceeded. SCAN returns an integer value indicating the number of bytes examined.

The scan length may be positive or negative. If the scan length is negative, SCAN proceeds backwards (i.e., towards lower addresses) from the starting address; otherwise, SCAN proceeds forward through memory. SCAN returns the zero-based offset from the starting address to the first occurrence of the target string. This value is negative when scanning backwards and positive when scanning forwards. If the scan terminates without finding the target string, the scan length is returned as the value of the function.

Example of SCAN:

```
program SCANdemo;
var farkle: string;
begin
  farkle := '.....the pterac is a member of the USCD family';
  writeln(scan(-26, = ':', farkle[30])); {writes -26}
```

```
    writeln(scan(100, <>'.', farkle[1] )); {writes 5  }
    writeln(scan(15 , = ' ', farkle[1] )); {writes 8  }
  end.
```

3.9 Device I/O

UCSD Pascal provides the intrinsics UNITREAD, UNITWRITE, UNITCLEAR, UNITSTATUS, UNITBUSY and UNITWAIT for accessing I/O devices. These intrinsics comprise the I/O level known as "Unit I/O"; this is the lowest level of I/O available to the system and must be used with care. Unit I/O is the p-System's fastest means of performing I/O. Further, the programmer may specify control parameters when using Unit I/O which are not available with standard I/O mechanisms. UNITREAD and UNITWRITE are described in section 3.9.1. UNITCLEAR, UNITBUSY and UNITWAIT are described in section 3.9.2. UNITSTATUS is described in section 3.9.3.

The primary argument to the Unit I/O intrinsics is the unit number, specifying an I/O device. Unit numbers and device assignments are system, and occasionally, installation dependent. Consult your System Installation Guide.

NOTE: The Version IV I/O redirection feature does not affect I/O operations performed using the Unit I/O intrinsics.

NOTE: The compiler does not generate I/O checks (section 5.0.6) after calls to Unit I/O intrinsics; I/O checks must be explicitly performed by examining the I/O completion status after every operation. (I/O completion status is examined with the IORESULT intrinsic –see section 3.11.9 for details.)

NOTE: The system routines implementing Unit I/O are protected from task contention.

3.9.1 UNITREAD and UNITWRITE

The UNITREAD and UNITWRITE intrinsics (in most cases) transfer data between memory and an I/O device. They accept a unit number, I/O buffer, byte count, and two optional parameters: a block number and a control word. The I/O buffer is specified by either an indexed or an unindexed variable name (e.g., `Arr[Index]`or `Arr`.) It is the source (for a

UNITWRITE) or the destination (for a UNITREAD) memory location from/to which the I/O will take place. The byte count is an unsigned integer in the range 0..65535, indicating the number of bytes to be transferred. The block number is a signed integer used in I/O involving block-structured devices; it indicates the starting block involved in the I/O. (A block contains 512 bytes.) The default value for the block number is zero, which indicates the first block on the unit. The control word specifies special processing options, such as special I/O or special character expansion; its default value is zero. The syntax for UNITREAD and UNITWRITE is described in sections 4.44 and 4.47. Their semantics are device dependent and are described in Appendix D.

WARNING: The most common results of incorrect use of UNITREAD and UNITWRITE are damaged disk files and/or directories, and program crashes caused by overrunning data buffers on read operations. No range checking is performed on accesses to the I/O buffer.

WARNING: Most UCSD Pascal implementations require that the I/O buffer start on a word boundary when performing disk operations. It is recommended that, to insure program portability, an I/O buffer start on a word boundary in all situations. Special care must be taken when using a byte array (e.g., a packed array of char) as an I/O source or destination.

NOTE: Array indices are treated as signed integers. Use of an index whose value is less than the I/O buffer's declared lower bound may yield unexpected or fatal results.

Unpacked variables of type CHAR occupy a full word. On some processors (e.g., the PDP/LSI-11, Z80, 8088), the character value is stored in the low-order byte. On others (e.g., the 68000 and TI-9900) it is stored in the high-order byte. A UNITREAD for one byte into a variable of type CHAR always fills the low-order byte. Thus, Unit I/O on character variables may not work as expected on some machines. In general, a safe procedure is to always use packed arrays of char with Unit I/O.

Further problems may arise, however, when using a UNITREAD for one byte into a CHAR variable, since the UNITREAD operation does not affect the high-order byte of the variable. The character will be displayed properly if written to the screen, but it will not compare properly to other characters since the entire word is considered in a character comparison. A CHAR variable used with a one-byte UNITREAD call should be initialized to any character value prior to the UNITREAD call. This sets the high-order byte of the variable to zero; the variable may then be correctly compared to other characters.

NOTE: Unit I/O intrinsics are used in a few cases for system actions unrelated to device I/O. For example, some implementations permit setting and monitoring the system clock via unit I/O; unit I/O may also be used to access memory above 64kb or to define various hardware characteristics. Consult your system documentation for details.

Example of UNITREAD and UNITWRITE:

```
program UnitIODemo;
var buff: packed array[0..2047] of char;
    ch: char;

  procedure putline(msg: string);
  var cr: packed array[0..0] of char;
  begin
    if length(msg) > 0 then
      unitwrite(2,msg[1],length(msg));
    cr[0] := chr(13);
    unitwrite(2,cr,1);
  end {putconsole};

  procedure getkey(var key: char);
  begin
    key := ' ';
    unitread(2,key,1);
                    {does not work on 68000's and TI-9900's}
  end {getkey};

begin
  putline('*** Screen Garbage Generator ***');
  putline('');
  putline('      G(arbage E(xit ');
  repeat
    getkey(ch);
  until ch in ['g','G','e','E'];
  if ch in ['e','E'] then exit(program);
  unitread(4,buff,2048,2);
  if ioresult <> 0 then
    begin
      putline('>>> I/O error detected');
      exit(program);
    end;
  unitwrite(1,buff,2048);
  putline('That''s all, folks...');
end.
```

All keyboard input and screen output in this program are done using Unit I/O. The procedure putline uses UNITWRITE to direct a message string to the system terminal device, which is usually device #1 or #2. Unlike WRITELN, no carriage return is appended when UNITWRITE is used, so it is necessary to send a separate CHR(13) to the screen (13 is the ASCII code for carriage return).

Procedure `getkey` uses UNITREAD to accept a single character input from the system terminal device. Note that the variable key is initialized to some value before the UNITREAD is performed. This enables the REPEAT/UNTIL loop in the main program to properly compare key to the character values specified in the set of the UNTIL clause. Note that a UNITREAD from unit #2 is non-echoing.

The main program attempts to UNITREAD 2048 bytes from block 2 of unit #4 (usually diskette drive 0) into the 2048 bytes of the array buff. If unit #4 is online and the read is successful, the bytes are written to the screen. Since no I/O checking is performed on Unit I/O calls, the programmer must always explicitly check IORESULT when using Unit IO; in this case, if the value of IORESULT is non-zero (indicating an I/O error) the program is designed to terminate with an error message.

3.9.2 UNITCLEAR, UNITBUSY and UNITWAIT

UNITCLEAR, UNITBUSY and UNITWAIT accept a unit number as a parameter. Their syntax is described in sections 4.43, 4.42 and 4.46. Their semantics are device dependent and are described in Appendix D.

The UNITCLEAR procedure resets and/or initializes I/O devices. In the case of serial input devices it usually clears the type-ahead buffer. The value of IORESULT after a UNITCLEAR call is often used to test the existence of the device; it should be zero if the device is present.

The UNITBUSY function is used to poll the status of an I/O device. It returns TRUE if the device has not completed a pending I/O; otherwise it returns FALSE. When it is used on a serial input device, it returns TRUE if no character has been received; otherwise it returns FALSE. UNITBUSY is not implemented on most UCSD Pascal systems; it always returns FALSE on such systems.

The UNITSTATUS procedure returns more complete information than UNITBUSY (see section 3.9.3).

The UNITWAIT procedure causes the current task to suspend activity until the specified unit has completed any I/O operation in progress. The task does not, however, relinquish control of the processor. UNITWAIT is not implemented on most UCSD Pascal systems; it has no effect when executed on such systems.

Example of UNITCLEAR and UNITBUSY:

```
program serialdemo;
var buff: packed array[0..0] of char;
begin
  unitclear(2);          {clear keyboard type-ahead}
  unitread(2,buff,1,,1);
  while unitbusy(2) do
```

```
    writeln('please type a character:');
  writeln ('character received: ', buff);
end.
```

On systems which implement UNITBUSY (i.e., the PDP/LSI-11) the execution of the UNITREAD call will cause UNITBUSY(2) to return TRUE until a character is typed. Thus, the WHILE loop continuously produces the message 'please type a character'. When a key is struck, a character is read into `buff` and UNITBUSY(2) becomes FALSE. The WHILE loop ends and the character is verified with the 'character received' message. Specifying asynchronous mode for the UNITREAD via the fifth parameter allows the program to continue to execute (and encounter the UNITBUSY test) as the UNITREAD proceeds.

3.9.3 UNITSTATUS

The UNITSTATUS procedure accepts a unit number, a status record and an integer as parameters. It returns status information on the specified device. The format of the status record depends on the device being polled; the status record may be of any type, but should occupy at least 30 words. The integer parameter should be zero for output status information and one for input status information.

UNITSTATUS is not available for all implementations. Where it is available the format of the status record may vary. Consult your system documentation for details.

The format of a status record for a serial device is:

```
SerialStatus = record
                 CharsQueued    : integer;
                 Filler         : array [1..29] of integer;
               end; {SerialStatus}
```

`CharsQueued` gives the number of characters currently available for input or output, as specified. The `Filler` words are either system-specific or reserved for future expansion.

The format of a status record for a block-structured device is:

```
BlockedStatus = record
                  Unused       : integer;
                  BytesSector  : integer;
                  SectorsTrack : integer;
                  TracksDisk   : integer;
                  Filler       : array [1..26] of integer;
                end; {BlockedStatus}
```

`BytesSector` gives the number of bytes per sector on the device as of the last time it was accessed. `SectorsTrack` contains the number of sectors per track. `TracksDisk` contains the number of tracks per disk. The

Fillerwords are either system-specific or reserved for future expansion.

NOTE: For some implementations, the value of IORESULT returned by the UNITSTATUS intrinsic reflects the presence of device handlers for the device polled, rather than device readiness. Consult your system documentation for specifics.

3.10 Inline Machine Code

UCSD Pascal provides the P_MACHINE intrinsic for generating in-line machine code within Pascal programs. In-line machine code is used for programming low level operations which cannot be expressed efficiently (if at all) in the Pascal language.

WARNING: P_MACHINE is the lowest level intrinsic in UCSD Pascal. Its use requires familiarity with the p-machine instruction set (see your Architecture Guide for details) and extreme care in specifying code sequences. Incorrect use of the P_MACHINE intrinsic may lead to bizarre and untraceable effects including data destruction and system crashes.

The P_MACHINE intrinsic permits the programmer to set up the stack to any desired configuration and then operate on it directly via p-code.

The P_MACHINE intrinsic accepts a series of any number of arguments. Multiple arguments are separated by commas. An argument is either an expression,an address reference or code.

An expression or address argument causes the compiler to generate code which, during execution, places the value of the expression or address onto the stack. A code argument causes the compiler to place a p-code instruction directly into the stream of compiler-emitted p-code instructions.

An expression is any valid Pascal expression enclosed in parentheses. The compiler generates code to evaluate the expression and leave the result on the stack.

An address reference is any valid Pascal variable reference preceded by the character '^'. The compiler generates code to leave the address of the specified variable on the stack.

A code argument consists of a constant value or constant identifier denoting an integer between 0 and 255. The compiler emits a single byte in the code with the specified value. Values outside of this range (e.g., signed constants) have only their least significant byte emitted. Code arguments are used for emitting p-code instructions and instruction operands.

A typical usage of P_MACHINE would be to first specify expressions and addresses to set up the stack, then specify code(s) for one or more instructions and operands to operate on the stack.

Example of P_MACHINE:

```
program PcodeDemo;
const STRL = 244;
type complex = record re,im: real end;
     vector =  array [0..10] of complex;
var  speed: ↑vector;
     x: real;
 begin
   new(speed);
   x := 3.14159;
   pmachine(↑speed↑[7].re, (x), STRL, 4);
   writeln('result is: ', speed↑[7].re);
 end {PcodeDemo}.
```

The STRL instruction stores a real value on the stack into a specified memory location. The real value is on the top of the stack; the destination address immediately precedes the real on the stack.

In this program, the first argument to the P_MACHINE intrinsic is an address, as denoted by the leading ^. Code is generated which places the address of `speed↑[7].re` on the stack. The next argument is an expression as denoted by the enclosing parenthesis. Code is generated which will push the value of `x` onto the stack. This will be the state of the stack at the time the STRL instruction is executed, since the instruction itself follows as the next argument to the P_MACHINE intrinsic. It is expressed as the previously defined constant identifier STRL. The op-code value 244 is therefore placed directly into the p-code.

This code follows the p-code that accomplishs the `x :=` assignment statement. The equivalent Pascal code would be `speed↑[7].re := x;`.

Some typical P_MACHINE operations:

```
const   STO = 196;       { store indirect    }
        IXA = 215;       { index array       }
        SIND0 = 120;     { load indirect     }
        BNOT = 159;      { boolean negation  }
        LEUSW = 180;     { unsigned <=       }
        GEUSW = 181;     { unsigned >=       }

var     i,j,index: integer;
        b: boolean;
        p: ↑integer;
        pb: ↑boolean;
        a: array[0..0] of integer;

{ It is necessary to reference the absolute
  address FC24 as an integer }
P_MACHINE ( ↑p, (-988), STO);
{ p↑ will reference memory address
```

```
    FC24 hex (= -988 decimal) }

  { It is necessary to reference the word "index" bytes
                                into array a as an integer }
  P_MACHINE ( ↑p, ↑a, (index), IXA, 1, STO);
  { p↑ will reference a[index] }

  { It is necessary to treat a boolean as an integer }
  P_MACHINE ( ↑p, (pb), STO);
  { p := pb; — an otherwise forbidden operation }

  { It is necessary to treat an address as an integer }
  P_MACHINE ( ↑i, (p), STO);
  { i := p; — an otherwise forbidden operation
    (but could legally be expressed in Pascal as
    i := ord(p);). Reversing i and p in the
    P_MACHINE above will allow the integer to
    be placed back into the pointer }

  { An illustration of indirect loading }
  P_MACHINE ( ↑i, (p), SIND0, STO);
  { i := p↑; —a perfectly legal way of
    doing the same thing }

  { Illustration }
  P_MACHINE ( ↑b, (i), (j), LEUSW, STO);
  { b := i <= j; (unsigned) — legal }

  { Illustration }
  P_MACHINE ( ↑b, (i), (j), GEUSW, BNOT, STO);
  { b := i < j; (unsigned) — legal }
```

3.11 Miscellaneous Extensions

This section describes miscellaneous extensions to standard Pascal. Sections 3.11.1 and 3.11.2 describe alterations of the syntax rules for identifiers and declaration parts respectively. Section 3.11.3 describes extension of the standard function ORD to perform pointer to integer type conversion.

The remaining extensions are the following UCSD intrinsics: GOTOXY for console cursor positioning (section 3.11.4), TIME for reading the system clock (section 3.11.5), PWROFTEN for real powers of ten (section 3.11.6), ATAN as an alternative name for the standard function ARCTAN (section 3.11.7), EXIT for terminating procedures or programs (section 3.11.8), IORESULT for checking the system I/O completion status (section 3.11.9), MEMAVAIL for checking the amount of unused memory (section 3.11.10), HALT for invoking the system monitor (section 3.11.11), and the compiler intrinsics TREESEARCH and IDSEARCH (section 3.11.12 and 3.11.13). Section 3.11.14 discusses a FOR control variable anomaly.

3.11.1 Identifiers

Identifiers in UCSD Pascal may contain the underscore character ”_”. Occurrences of ”_” embedded within identifiers are ignored by the compiler; thus, the identifiers ”procnum” and ”proc_num” are equivalent. Identifiers are case-insensitive.

NOTE: Identifiers are significant to 8 characters, excluding underscore characters.

WARNING: Although they contribute to program readability, long variable names should be used carefully in UCSD Pascal, as it is disconcertingly easy for two ”different” long variable names to map into the same identifier because of the eight-character rule. Identifier aliases can cause mysterious compiler syntaxerrors and/or elusive program bugs.

The following identifiers are equivalent in UCSD Pascal:

```
identifier
i_dent_i_fi_er
Identifier
IDENTIFI
I_dent_I_fier
identifier_of_sparse_matrix_
```

3.11.2 Declaration Parts

Suites of related programs often must share a common group of label, constant, type, variable and procedure declarations. In UCSD Pascal, this can be done by placing the source code defining the group in a separate include file (section 5.0.2), and having each program in the suite include the declarations into its declaration part. (Another way is to put the declarations in a UNIT; see section 3.2 for details).

Standard Pascal restricts the ordering of declarations in a declaration part so that labels are declared before constants, constants are declared before types, and so on. Because of these restrictions an include file containing a set of related declarations would not compile successfully when included in a host program's declaration part, as the declaration order would be violated (e.g., given an include file containing label, constant, type and variable declarations, what is the proper location within the host program's uses, constant, type and variable declarations for the include file directive?). Thus, UCSD Pascal relaxes the restrictions on declaration order for include files appearing in declaration parts.

In particular, it is permissible to place the include file anywhere in the host program prior to the first procedure body. In fact, it is even permissible to arrange the const, type and var declarations in the include file itself in any order. (The procedure bodies of the include file must follow any declarations.) Of course, a declaration referencing another identifier will cause a compiler error if the other identifier is not declared first. Note that the usual Pascal declaration sequences are enforced for declarations appearing in the *program* file.

NOTE: Files containing label, constant, type, or variable declarations may not be included after a procedure body; however, they may be included after forward declarations.

Example of included declarations:

```
*** The include file (named INC1.TEXT):

const
  H = 'Hi, guys! This is Eddy, your shipboard computer!';
type  car = record
              make: string;
              license: integer;
            end;
var   c, a: car;

  procedure rice(r: car);
  begin
    a := c;
    writeln(H);
  end;


*** The host program:

program margorp;
const N = 5;
var i,j,k: integer;

{$I INC1.TEXT — If the text of the include file
    were physically present instead of included, this
    program would bomb miserably during compile.
    The declarations would be out of order! }

  procedure useless;
  begin
    { Just to show that the include file
      precedes the procedure bodies }
  end;

begin
  with c do
    begin make := 'Edsel'; license := 10101 end;
  writeln(H);
```

```
    for i := 1 to N do
      begin writeln('and again...'); rice(c) end;
  end {margorp}.
```

3.11.3 Pointer Type Conversion and Comparison

In UCSD Pascal, the standard procedure ORD is extended to accept pointer types. This extension should only be used in machine-specific tasks requiring pointer-to-integer type conversion. See section 3.10 for information on integer-to-pointer conversions. See sections 6.7 and 6.8 for examples of such type conversions.

In UCSD Pascal, comparisons of pointers using the <, <=, > and >= operators are allowed in addition to the = and <> operators allowed in Standard Pascal. These operators are evaluated as 16-bit unsigned comparisons are meant for use in systems programs performing memory management.

NOTE: Pointer values on most machines may be considered scalar values (unsigned 16-bit) in the range 0..65535, which correspond to memory addresses. When considered as integers (signed 16-bit) as returned by ORD(<pointertype>) they are in the range -32768..32767. See section 6.2 for more information.

3.11.4 Screen Control

The UCSD intrinsic GOTOXY provides terminal-independent X-Y coordinate cursor positioning. GOTOXY may be used in conjunction with READ, READLN, WRITE and WRITELN to create formatted screen displays and prompt lines.

GOTOXY is a terminal-dependent procedure, and thus must be written and bound into the operating system by the user. Consult your System Installation Guide for details. Once the binding process is complete, however, GOTOXY may be used as any other intrinsic.

NOTE: A common error when using GOTOXY is to assume that the first parameter refers to row number and the second to column number. In reality, the first parameter is the column number (i.e.,the x coordinate on the screen) while the second is the row number (y coordinate on the screen). The valid range for these values is defined in the GOTOXY procedure itself, and in SYSTEM.MISCINFO, a file of system attributes which is also set up when the p-System is installed. For a typical 24 x 80 character screen, the first parameter will range in value from 0 to 79; the second from 0 to 23.

Example of GOTOXY:

```
program SControl;
type horz = 0..79;
     vert = 0..23;

  procedure putline(x: horz; y: vert; line:string);
  var len: integer;
  begin
    gotoxy(x,y);
    write(line);
    gotoxy(x,y+1);
    for len := 1 to length(line) do write('-');
  end;

begin
  putline(37, 2,'North');
  putline(76,12,'East' );
  putline(37,21,'South');
  putline(0 ,12,'West' );
end {SControl}.
```

This program uses procedure `putline` to write each of the four directions at the appropriate point on the screen. `Putline` goes to the specified set of coordinates and writes the direction. It then goes to the line below the direction `(y+1)` and underlines it.

3.11.5 Clock Access

The UCSD intrinsic TIME provides access to the system clock. The clock is defined as a 32-bit unsigned integer incremented every 60th of a second (a "tick"). The clock value is returned in a pair of integers passed to TIME as variable parameters, where the first integer receives the high-order 16 bits of the 32-bit integer, and the second integer receives the low-order 16 bits. Note that the integers themselves contain unsigned values (see section 6.2).

TIME is normally used to time intervals. It does not return the value of a real-time clock.

Example of TIME:

```
program timer;

const
  twoto16th = 65536.0;
  beep = 7;
  hellfreezesover = false;
var
  high1, low1,
  high2, low2,
  killtime           : integer;
  presentime,
  prevtime, interval : real;

function realtime (hi : integer; lo : integer) : real;
  var r : real;
  begin
    r := lo;
    if r < 0.0
      then r := r + twoto16th;
    r := (hi * twoto16th) + r;
    realtime := r;
  end;

begin
  writeln ('Infinite Interval Timer');
  write   ('What is your interval? (input a real) ');
  readln  (interval);
  interval := interval * 60.0;  { seconds —> ticks }

  { Is line-time clock on? }
  time (high1, low1);
  for killtime := 1 to 1000 do {nothing};
  time (high2, low2);
  if (high1 = high2) and (low1 = low2)
    then begin
      writeln (
    'Can''t time intervals without a clock. Turn it on!');
      while (low2 = low1) do time (high2, low2);
    end;

  { Time intervals forever }
  prevtime := realtime (high2, low2);
  repeat
    repeat
      time (high1, low1);
      presentime := realtime (high1, low1);
    until (presentime - prevtime) > interval;
    prevtime := presentime;
    write (chr(beep));
  until hellfreezesover;
end.
```

Program `timer` causes the terminal to beep at the user at the end of every interval of a specified number of seconds. The user is prompted for the interval, which is multiplied by 60 to convert it from seconds to ticks. The program then assures that the clock is functioning. It samples the TIME, kills a bit of time, and samples the time again. If the values returned by TIME are the same in both samples, the clock is assumed to be off. A message is produced, and the program loops until a different value is returned by TIME (i.e., until the clock is turned on).

Since it is difficult to work with a 32-bit integer stored in two parts, the program uses procedure `realtime` to convert the number returned by TIME into a single real value. First, the low-order integer is examined. If it is greater than or equal to zero it is assigned directly to the real `r`. But if it is less than zero (as a two's complement value) it must first be converted to an unsigned 16-bit integer. This is accomplished by adding it to 2 to the 16th power, or 65536. The high-order integer is then scaled upwards to its true value as the leftmost 16 bits of a 32-bit number by shifting it to the left 16 times. This is accomplished by multiplying it by 2 to the 16th. The resulting product is added to the real containing the low-order component, and the conversion to a single real value is complete.

The interval timing then begins. The inner loop continues to repeat until the time between the previous beep and the present is the specified interval. Another beep is produced, the time of the previous beep is updated, and the process continues.

NOTE: The TIME intrinsic normally returns 0 under p-System implementations which do not permit access to a system clock.

3.11.6 Powers of Ten

The UCSD intrinsic PWROFTEN (short for "power of ten") accepts an integer argument and returns a real result equal to ten raised to the power of the argument.

The maximum size of the integer argument depends on the representation of reals utilized by a particular interpreter. Consult your documentation for details.

NOTE: Arguments less than 0 and arguments that are too large cause the execution error "Floating point error".

Example of PWROFTEN:

```
program powers;
var i: integer;
begin
  repeat
    write('enter arg: ');
    readln(i);
    writeln('arg is ',i,' result is ',pwroften(i));
  until i = 0;
end {powers}.
```

3.11.7 Arctangent Synonym

Standard Pascal defines ARCTAN as the standard function for the arctangent function. In UCSD Pascal, both ARCTAN and ATAN denote the arctangent function.

3.11.8 Procedure Termination

The UCSD intrinsic EXIT accepts a procedure, function or program identifier as an argument. It causes execution to continue at the end of the block named by the identifier. The simplest case of EXIT occurs when the identifier denotes the current procedure; EXIT jumps to the end of the current procedure. If the EXIT argument specifies a routine at an outer level, all routines on the call chain between the current routine and the specified routine are also terminated. The entire program may be terminated by calling EXIT with either the program identifier or the reserved word PROGRAM.

NOTE: If EXIT specifies a recursively invoked procedure, only the most recent invocation is terminated. A workaround is possible in cases where it is desired to terminate all invocations of the recursive procedure: enclose the recursive procedure in a dummy procedure which does nothing but invoke the recursive procedure. To EXIT the recursive procedure, simply EXIT the enclosing dummy procedure.

```
program TestRecurs;

procedure DumRecurs;

procedure Recurs;
begin {Recurs}
  ...
  if {time to get out}
    then EXIT(DumRecurs)
    else Recurs;
end {Recurs};

begin {DumRecurs}
  Recurs;
end {DumRecurs};

begin {TestRecurs}
  ...
  DumRecurs;
  ...
end.
```

NOTE: A process may not EXIT out of its block. Attempts to do so result in the termination of the process.

NOTE: `EXIT(PROGRAM)` is the only legal form of the EXIT statement in the initialization or termination sections of a unit. In the initialization section it causes control to pass directly to the termination section of that unit. In the termination section it ends the program, unless termination code from other units is pending. EXIT with a unit identifier is meaningless.

NOTE: GOTO statements naming a label outside of the current block (called "out-of-block" GOTOs) are not implemented in UCSD Pascal. The EXIT intrinsic is used to provide an alternative (albeit limited) form of out-of-block GOTO.

NOTE: A CLOSE(<file>, NORMAL) is performed on all files local to a procedure terminated by a call to EXIT.

WARNING: When EXIT is used to terminate a function, the function value must have been assigned beforehand; otherwise, the function returns an undefined value.

Example of EXIT:

```
program exitdemo;
var num: integer;

  procedure readNatural(var int: integer);
  var ch: char;

    procedure blowout(errmsg: string);
    var ch: char;
    begin
      writeln;
      writeln('>>>Error: ',errmsg);
      write('   type <space> to continue, "!" to escape');
      repeat read(keyboard,ch) until ch in [' ','!'];
      writeln(ch);
      if ch = '!' then exit(program)
      else exit (readNatural);
    end {blowout};

  begin
    int := 0;
    repeat
      read(ch);
      if not (ch in ['0'..'9',' ']) then
        blowout('Input format');
        if ch = ' ' then exit(readNatural);
        if (int > maxint div 10) or ((int = maxint div 10)
           and (ord(ch) - ord('0') > maxint mod 10)) then
             blowout('Integer too large');
        int := int * 10 + ord(ch) - ord('0');
      until false;
    end {readNatural};

  begin
    repeat
      write(' enter nonnegative number (17 terminates): ');
      readNatural(num);
      writeln(' number entered is: ',num);
    until num = 17;
  end {exitdemo}.
```

The program calls procedure `ReadNatural` to read a string of digit characters and convert them to an integer. If `ReadNatural` encounters an illegal character or a value about to become out of range it calls on procedure `blowout` to issue an error message, and give the user the opportunity to read more natural numbers or quit. If the user opts to quit, an `EXIT(PROGRAM)` is issued. If the user wishes to continue an `EXIT(ReadNatural)` is issued. This has the effect of returning control to the writeln statement in the main program.

Of note here is the technique used to convert the characters to an integer. An integer variable is initialized to zero. As the characters are input and validated they are converted from their ASCII representations to numerical values in the range 0..9 by subtracting ORD(0) from them. They are added to the integer, which is scaled upwards (shifted to the left one place) by multiplying it by 10.

Before the latest character input is added to the integer, however, the integer is checked to assure that the upcoming increase in its value will not cause it to exceed MAXINT. Of course, once the character is added to the integer, it will no longer be possible to detect the overflow; the value of the integer will appear to be negative. (The p-System does not check for integer overflow.)

3.11.9 I/O Completion Status

The UCSD intrinsic IORESULT returns an integer result containing the current value of the system I/O completion status. The status is updated after every I/O operation (including file operations). Calling IORESULT is usually unnecessary, as the compiler automatically generates I/O checks after all I/O operations with the exception of Unit I/O operations. If, however, I/O checks are suppressed (using the $I- compile option – see section 5.0.6), the value returned by IORESULT should be explicitly checked after each I/O operation to prevent I/O errors from causing program errors.

IORESULT is used in programs which substitute their own error checking and recovery for the system's error handling facilities. If a program does its own error checking only in one particular section, I/O checking must be explicitly restored afterwards using the $I+ or $I^ compiler options.

NOTE: Appendix A lists the standard I/O result values and their definitions. Appendix C lists conditions causing bad I/O results.

NOTE: The low-level Unit I/O intrinsics (section 3.9) always require explicit I/O checks. The compiler does not generate I/O checks after occurrences of these intrinsics.

WARNING: Because the I/O completion status word is reset after every I/O operation, care must be taken to preserve I/O result values between their detection and subsequent I/O operations. In the following example, the bad I/O result generated by the reset is inadvertently obliterated by the ensuing string write before it reaches the console:

```
{$I-}
program inoperative;
var f: file;
begin
  reset(f,'nonexistent.file.text');
  writeln('I/O result after file open is ',ioresult);
end {inoperative}.
```

The program will erroneously report that the I/O result after attempting to open the nonexistent file is 0! It is really reporting that the I/O result of the writeln of the message string itself is 0. The programmer should have set an integer variable to the value of IORESULT immediately after the reset, then written the value of that variable along with the message.

Example of IORESULT:

```
program IOdemo;
var num: integer;

  procedure getnum(prompt: string; var int: integer);
  begin
    {$I-}        { suppress I/O checks }
    repeat
      write(prompt);
      readln(int);
    until ioresult = 0;
```

```
      {$I+}
    end {getnum};

  begin
  repeat
    getnum('Enter number (-1 terminates): ',num);
    writeln(' number returned is: ',num);
  until num = -1;
  end {IOdemo}.
```

This program demonstrates how one can repeatedly prompt the user to input a proper value, rather than causing a runtime error when an improper value is supplied. The loop reading the integer will continue until the value of IORESULT is 0, indicating a valid integer value was input.

Section 6.9.3 contains another example using IORESULT.

3.11.10 Memory Available

The UCSD intrinsic MEMAVAIL returns the number of unused words in memory as an unsigned integer.

NOTE: Values returned by MEMAVAIL should be treated as 16-bit values in the range 0..65535; values in the range 32768..65535 are ordinarily considered negative numbers in the signed 16-bit representation used for integer types and operators. See section 6.2 for more information.

NOTE: Values returned by MEMAVAIL are best used on Version II systems, where they represent the number of words between the system stack and current heap. When using Version IV the VARAVAIL intrinsic produces more useful results. See section 3.5 for further information. (On Version IV systems, MEMAVAIL returns the number of words between the stack and the heap, exclusive of any memory allocated to an internal code pool.)

Example of MEMAVAIL:

```
  program memDisplay;
  begin
    writeln(' Integer memavail value = ',memavail);
  end {memDisplay}.
```

Section 6.4 contains other examples.

WARNING: When using Version IV of the p-System MEMAVAIL and VARAVAIL should be employed with extreme caution in multiprocessing programs and programs that use DISPOSE. In multiprocessing programs these intrinsics return the space between the process stack and the bottom of the process stack –a useless value for heap allocation calculations. Programs that use DISPOSE create "holes" of unused space in the heap; these intrinsics return a value that is the sum of the size of the largest such "hole" and the available stack size. Again, this value is useless for heap allocation calculations.

3.11.11 Programmed Halt

The UCSD intrinsic HALT causes the program to halt execution with the runtime error message "Programmed HALT". If the p-System Debugger had been activated it gains control and the programmer may use its facilities. Otherwise, typing <space> will re-initialize the system while typing <esc> will cause the program to resume execution at the instruction following the HALT.

Example of HALT:

```
program succinct;
begin
  halt;
end {succinct}.
```

3.11.12 Compiler Support - TREESEARCH

The compiler uses the UCSD intrinsics IDSEARCH and TREESEARCH for scanning identifiers and maintaining symbol trees respectively. IDSEARCH is a specialized intrinsic suited for use only in the construction of compilers. It is discussed in section 4.14 and in the next section. However, TREESEARCH performs a more generalized task, which will be described here.

These intrinsics perform high-level functions and would ordinarily not be implemented at the interpreter level. However, both intrinsics are processor-intensive, so providing them at the interpreter level – which permits them to operate at machine-language speeds – allows the compiler to operate much more quickly than it would otherwise. Of course, the same is true for any other program making use of these intrinsics.

TREESEARCH manages binary trees ordered by the contents of an 8-character field. A node in the tree must be structured as follows (field names may vary):

```
type nodeptr = ↑node;
     node = record
              name: packed array [1..8] of char;
```

```
  right_link: nodeptr;
  left_link: nodeptr;
  ...
  { user-defined record fields }
  ...
end {node};
```

TREESEARCH accepts a tree root pointer, node pointer and an 8-character packed array as arguments. It returns an integer as a function result, and also assigns a value to the node pointer parameter. If a node in the tree matches the array argument, TREESEARCH returns the value 0 as a function result; the node pointer is set to the node. If no node in the tree matches the array argument, TREESEARCH returns either 1 or -1 and the node pointer is set to the last node searched. 1 indicates that the array argument is greater than the value of the last node and should be inserted on its right link; -1 indicates that the argument is less than the last node and should be inserted on its left link.

The programmer must assure, when building the tree, that the pointers of the leaf nodes are initialized to NIL.

NOTE: Trees should be constructed for the TREESEARCH intrinsic so that visiting the nodes in lexicographical order with respect to name field requires either right (post-order) traversals or left (pre-order) traversals of the tree. The particular traversal may vary in the various UCSD Pascal implementations.

Example of TREESEARCH:

```
program tree_demo;
type alpha = packed array [1..8] of char;
     nodeptr = ↑node;
     node = record
              name: alpha;
              llink, rlink: nodeptr;
              value: integer;
            end {node};
var  root: nodeptr;
     cmd: char;

  procedure get_name(var name: alpha);
  var s: string;
      cnt: integer;
  begin
    readln(s);
    name := '        ';
    for cnt := 1 to length(s) do
      if cnt <= 8 then name[cnt] := s[cnt];
  end {get_name};

  procedure find_node;
  var entry: nodeptr;
```

```
      search: alpha;
  begin
    write(' Enter name: ');
    get_name(search);
    if treesearch(root,entry,search) <> 0 then
      writeln('  Entry not found')
    else with entry↑ do
      writeln('  Name: ',name,' Value: ',value);
  end {find_node};

  procedure print_nodes(tree: nodeptr);
  begin {print nodes in ascending lexigraphic order}
    if tree↑.rlink <> nil then
      print_nodes(tree↑.rlink);
    with tree↑ do
      writeln('  Name: ',name,' Value: ',value);
    if tree↑.llink <> nil then
      print_nodes(tree↑.llink);
  end {print_nodes};

  procedure add_node;
  var new_node, entry: nodeptr;
      result, new_val: integer;
      new_name: alpha;
  begin
    write(' Enter name: ');
    get_name(new_name);
    result := treesearch(root,entry,new_name);
    if result = 0 then
      writeln('  Entry already exists')
    else
      begin
        new(new_node);
        with new_node↑ do
          begin
            write(' Enter value: ');
            readln(value);
            name := new_name;
            llink := nil; rlink := nil;
            if result = 1 then
              entry↑.rlink := new_node
            else
              entry↑.llink := new_node
          end;
      end {else};
  end {add_node};

begin
  new(root);
  with root↑ do
    begin
      name := '        '; value := 0;
      llink := nil; rlink := nil;
    end;
  repeat
    write('A(dd node F(ind node P(rint node Q(uit');
      read(keyboard,cmd); writeln(cmd);
      case cmd of
        'a': add_node;
        'f': find_node;
        'p': print_node(root);
      end;
    until cmd = 'q';
  end {tree_demo}.
```

This program uses TREESEARCH to build, search and print a binary tree. Procedure `add_node` uses TREESEARCH to determine if the name to be added is already in the tree. If not, a new node is allocated and its pointers are set to NIL. The node pointer variable parameter supplied by TREESEARCH indicates the father node of the new one. The integer value returned by TREESEARCH determines whether the new node is to be a left son or a right son. Procedure `find_node` uses TREESEARCH to find the desired node. Procedure `print_nodes` recursively performs a post-order traversal of the entire tree, printing the information in each node. The main program initializes the root node and prompts the user to choose one of the provided functions.

3.11.13 Compiler Support - IDSEARCH

The IDSEARCH intrinsic is a special-purpose procedure designed to speed up the the search for valid Pascal identifiers in a Pascal source program. It was designed for use in the UCSD Pascal compiler, but it can be useful in any program that accepts Pascal source text as input, such as cross-reference generators, pre-processors and the like.

IDSEARCH accepts two parameters, `INX` and `BUFFER`. `BUFFER` is normally a packed array[0..1023] of char, presumed to contain part of a Pascal source program. `INX` is an index into `BUFFER`; it must be preset to point to the first character of a Pascal identifier –an upper or lower case letter. IDSEARCH identifies the identifier found, the category of the identifier (e.g., an operator, a user-defined identifier, etc.), and, if the identifier is an operator, which operator it is. `INX` is left pointing to the last character in the identifier.

IDSEARCH is a "dirty" procedure. Notice that both parameters to IDSEARCH are input parameters; neither provides for the return of the information which IDSEARCH produces. This information is placed in the variables which are declared immediately following the declaration of the actual parameter corresponding to `INX`. IDSEARCH uses the address of `INX` (`INX` is a variable parameter so its address is known to IDSEARCH) to determine where to store these values. It is the programmer's responsibility to declare the output parameters so that they are of the appropriate type and in the appropriate position relative to the `INX` parameter.

The following variables must be of the type and in the sequence shown:

```
var
  index: integer;
       {the actual parameter corresponding to INX}
  sym  : symbol;
       {will contain the category of the identifer found}
  op   : operator;
       {if identifier is an operator, which one; else NOP}
  id   : alpha;
       {which identifier was found; first 8 characters}
```

Type `alpha` is a packed array[1..8] of char. Types `symbol` and `operator` are enumerated types which define the categories of keywords and valid operators, respectively. These are illustrated in the program below.

Type `symbol` defines categories of keywords; however, this type enumerates other symbols besides keywords since it is used by the UCSD Pascal compiler to recognize all valid Pascal symbols. The entire type must therefore be declared as shown; it should be understood, however, that IDSEARCH only returns values in `sym` that correspond to keywords. These are shown below in capital letters.

WARNING: As the Pascal language changes, so will the values enumerated in the `symbol` type. The version of `symbol` given below functions in Versions II through IV of the p-System. There were a number of new values added to `symbol` after Version II; these were added at the end of the enumeration, however, so the declaration below still works for Version II Pascal programs.

Example of IDSEARCH:

```
program match;
type
  symbol =
 (IDENT,comma,colon,semicolon,lparen,rparen,DOsy,TOsy,
  DOWNTOsy,ENDsy,UNTILsy,OFsy,THENsy,ELSEsy,becomes,lbrack,
  rbrack,arrow,period,BEGINsy,IFsy,CASEsy,REPEATsy,WHILEsy,
  FORsy,WITHsy,GOTOsy,LABELsy,CONSTsy,TYPEsy,VARsy,PROCsy,
  FUNCsy,PROGsy,FORWARDsy,intconst,realconst,stringconst,
  NOTsy,MULOP,ADDOP,RELOP,SETsy,PACKEDsy,ARRAYsy,RECORDsy,
  FILEsy,OTHERsy,longconst,USESsy,UNITsy,INTERsy,IMPLEsy,
  EXTERNLsy,SEPARATsy,qstar,PROCESsy);

{ IDENT is used for user-defined identifiers. PROGsy is
  used for SEGMENT as well as PROGRAM. MULOP designates
            DIV, AND and MOD }
  operator = (MUL,RDIV,ANDOP,IDIV,IMOD,PLUS,MINUS,OROP,
              LTOP,LEOP,GEOP,GTOP,NEOP,EQOP,INOP,NOOP);

  alpha    = packed array[1..8] of char;

var
  index: integer; {the order of these four}
  sym  : symbol;  {parameters is critical}
  op   : operator;
  id   : alpha;

  bal,
  blks : integer;
  src  : string;
  suf  : string[5];
  fyle : file;
  buf  : packed array[0..1023] of char;

function MoveWhenInBuffer(lim: integer): boolean;
var
  upper: integer;
begin
  upper := (lim * 512) - 1;
              { don't go higher than 511 if one block  }
  {$R-}       { or higher than 1023 if two blocks read }
  repeat
    index := succ(index);
  until ((buf[index] in ['A'..'Z', 'a'..'z']) or
         (index >= upper));
  {$R+}
  if index >= upper
    then MoveWhenInBuffer := FALSE
    else MoveWhenInBuffer := TRUE;
end;

begin
  bal := 0;
  suf := '     ';
  write('Your source file: ');    readln(src);
  reset(fyle, src);
  if length(src) > 5
    then suf := copy(src, length(src)-4, 5);
    if suf = '.TEXT' or suf = '.text'
      then blks := blockread(fyle, buf, 2, 2)
                 {skip 2-block text file header}
      else blks := blockread(fyle, buf, 2);

  while (blks > 0) do begin
    index := 0;
    while MoveWhenInBuffer(blks) do begin
      idsearch(index, buf);
      if sym = REPEATsy
        then bal := succ(bal)
        else if sym = UNTILsy
               then bal := pred(bal);
    end;
    blks := blockread(fyle, buf, 2);
  end;

  if bal = 0
    then writeln(
    'Congratulations, your REPEAT''s and UNTIL''s match!')
  else if bal < 0
    then writeln(
 'Oops, you have ',-bal,' more UNTIL''s than REPEAT''s!')
  else
    writeln(
 'Oops, you have ',bal,' more REPEAT''s than UNTIL''s!')
end.
```

This program determines if a specified Pascal source program has matching sets or REPEAT/UNTIL. Two blocks at a time are read into `buf`; the `index` is advanced by function `MoveWhenInBuffer` to the next character eligible to be the beginning of an identifier. `MoveWhenInBuffer` returns FALSE when it advances beyond the end of the current buffer, and the program continues to read from the source program until the end of the file. (The system editors and operating system cooperate to assure that no identifiers cross the two-block boundary.)

When `MoveInBuffer` returns TRUE the IDSEARCH procedure is invoked and the variable `sym` is examined. If it indicates a REPEAT was found, the variable `bal` is incremented by one (`bal` was initialized to zero). If `sym` indicates an UNTIL was found, `bal` is decremented by one. When the entire source file has been traversed, the value of `bal` indicates the balance of REPEAT's to UNTIL's in the source program.

Note that this program is not very clever regarding REPEAT's and UNTIL's embedded in comments!

3.11.14 FOR Control Variables

In Standard Pascal the control variable of a FOR loop must be declared locally in the procedure containing the FOR statement. This restriction has been lifted in UCSD Pascal. The control variable may be any free-standing integer or scalar variable (i.e., not part of a structured variable) at any scope level visible at the FOR statement. For example:

```
program lax;
var
  index: integer;

procedure UseVar;
begin
  for index := 0 to 5 do
    writeln('Only allowed in UCSD Pascal');
end;
```

```
begin
  UseVar;
end.
```

Additionally, Standard Pascal prohibits the modification of the control variable within a FOR loop. UCSD Pascal does not flag this as an error, although this practice is not recommended.

Chapter 4

UCSD INTRINSICS

Contents

4.0 ATAN 113
4.1 ATTACH 114
4.2 BLOCKREAD 114
4.3 BLOCKWRITE 115
4.4 CHAIN 116
4.5 CLOSE 117
4.6 CONCAT 118
4.7 COPY 118
4.8 DELETE 118
4.9 EXCEPTION 119
4.10 EXIT 119
4.11 FILLCHAR 120
4.12 GOTOXY 121
4.13 HALT 121
4.14 IDSEARCH 122
4.15 INSERT 122
4.16 IORESULT 122
4.17 LENGTH 123
4.18 MARK 123
4.19 MEMAVAIL 124
4.20 MEMLOCK 124

4.21 MEMSWAP 125
4.22 MOVELEFT 125
4.23 MOVERIGHT 126
4.24 OPENNEW 127
4.25 OPENOLD 127
4.26 P_MACHINE 127
4.27 POS 128
4.28 PWROFTEN 128
4.29 REDIRECT 129
4.30 RELEASE 129
4.31 RESET 130
4.32 REWRITE 131
4.33 SCAN 131
4.34 SEEK 132
4.35 SEMINIT 133
4.36 SIGNAL 133
4.37 SIZEOF 133
4.38 START 134
4.39 STR 134
4.40 TIME 135
4.41 TREESEARCH 135
4.42 UNITBUSY 136
4.43 UNITCLEAR 136
4.44 UNITREAD 136
4.45 UNITSTATUS 137
4.46 UNITWAIT 138
4.47 UNITWRITE 138
4.48 VARAVAIL 139
4.49 VARDISPOSE 139
4.50 VARNEW 140
4.51 WAIT 140

This chapter contains descriptions of all UCSD Pascal intrinsics, listed in alphabetic order. Most descriptions contain a reference to a related section in chapter 3, which describes the intrinsics in terms of the features they implement (and also presents programming examples). A number of intrinsics are described fully in this chapter. These include limited-use intrinsics such as IDSEARCH, and the intrinsics which make use of an operating system unit, such as CHAIN. Users unfamiliar with the UCSD Pascal intrinsics should peruse chapter 3 before reading this chapter.

With two exceptions, the identifiers chosen to denote UCSD Pascal intrinsics are distinct from those of the procedures defined in Standard Pascal. RESET and REWRITE are sufficiently altered to warrant inclusion in this section as UCSD Pascal intrinsics.

To illustrate the parameters accepted by each intrinsic, its function or procedure "heading" is provided. This heading resembles the way the intrinsic might actually have been declared when originally written.

NOTE: In order to completely specify the UCSD Pascal intrinsics, this section embellishes Pascal syntax with the metasymbols defined in section 1.3 and two special "type" identifiers. Metasymbols are used to indicate optional parameters ([<opt-param>]) and sequences of one or more parameters ({<param-seq>}). The special "type" identifiers indicate relaxed type checking on the corresponding parameter(s). The "type" UNIV denotes a universal type; formal parameters described as being of "type" UNIV accept actual parameters of any type. The "type" FILEID is compatible with all file types. These "type" identifiers are used for descriptive purposes only. They are not a part of the UCSD Pascal language.

NOTE: Most extensions described in this chapter are considered as a part of the UCSD Pascal base language as it exists in Version IV. Another class of extensions is available through the use of library modules. Units that allow extended file and directory management, screen control and other system-oriented functions are included or available as options with Version IV of the p-System. Some of these are described in chapter 8. Units are also available from a variety of vendors; they extend the language in a manner suited for a variety of application areas.

4.0 ATAN

Syntax:

```
function atan(X: real): real;
```

ATAN is equivalent to the standard procedure ARCTAN. See section 3.11.7 for more information.

4.1 ATTACH

Syntax:

```
procedure attach(var SEM    : semaphore;
                     VECTOR : integer);
```

ATTACH associates the semaphore variable SEM with the event (p-machine interrupt vector) specified by VECTOR; the p-machine signals SEM whenever the specified event occurs.

Events and their event numbers are defined in the I/O system. Certain event values are standard over most Version IV releases; others are system-dependent. Consult your Architecture Guide for details.

Only one semaphore may be attached to an event at a time; attaching a new semaphore to an event implicitly de-attaches the old semaphore. A semaphore may be de-attached from an event without attaching another semaphore by attaching NIL.

See section 3.0.2 for more information.

4.2 BLOCKREAD

Syntax:

```
function blockread(
    var F        : file;
    var BUFF     : univ;
        BLOCKS   : integer
     [; RELBLOCK : integer]) : integer;
```

BLOCKREAD attempts to read the number of blocks specified by BLOCKS from the file F into the variable BUFF. It returns the number of blocks actually read. If the number of blocks returned is less than the number of blocks requested, BLOCKREAD encountered either the end of the file or an I/O error while reading the data.

I/O checks are automatically generated for BLOCKREAD calls. If I/O checks are suppressed (with the $I- compiler option) IORESULT should be used to check the completion status after BLOCKREAD calls.

The optional parameter RELBLOCK applies only when reading from block-structured (disk) files. When specified, it indicates the block in the file where BLOCKREAD starts reading. The starting block is relative to the beginning of the disk file, with block 0 being the first block in the file.

In the absence of a RELBLOCK parameter, blocks are read from the file consecutively. The first BLOCKREAD after F is opened reads from block 0.

RELBLOCK is ignored when reading from serial devices.

On many machines, disk I/O may only occur to/from word-aligned memory addresses. Users must ensure that BUFF specifies a word-aligned address as the starting address of the buffer. This is most important when the starting buffer address is an indexed address in a packed array of char, since each element of the array occupies a single byte.

Users are responsible for not overrunning the buffer specified by BUFF since no range checking is provided when performing BLOCKREAD.

See section 3.3.4 for more information.

4.3 BLOCKWRITE

Syntax:

```
function blockwrite(
    var F        : file;
    var BUFF     : univ;
        BLOCKS   : integer
     [; RELBLOCK : integer]) : integer;
```

BLOCKWRITE attempts to write the number of blocks specified by BLOCKS from the variable BUFF to the file F. It returns the number of blocks actually written. If the number of blocks returned is less than the number of blocks requested, BLOCKWRITE encountered an I/O error while writing the data.

I/O checks are automatically generated for BLOCKWRITE calls. If I/O checks are suppressed, IORESULT should be used to check the completion status after BLOCKWRITE calls. Writing beyond the end of a file automatically extends the file if possible; otherwise, an I/O error is returned.

The optional parameter RELBLOCK applies only when writing to block-structured (disk) files. When specified, it indicates the block in the file where BLOCKWRITE starts writing. The starting block is relative to the front of the disk file, with block 0 being the first block in the file.

In the absence of a RELBLOCK parameter, blocks are written to the file consecutively. The first BLOCKWRITE after F is opened writes to block 0.

RELBLOCK is ignored when writing to serial devices.

On many machines, disk I/O may only occur to/from word-aligned memory addresses. Users must ensure that BUFF specifies a word-aligned address as the starting address of the buffer. This is most important when the starting buffer address is an indexed address in a packed array of char, since each element of the array occupies a single byte.

Users are responsible for not overrunning the buffer specified by BUFF since no range checking is provided when performing BLOCKWRITE. See section 3.3.4 for more information.

4.4 CHAIN

Syntax:

```
procedure chain (execution_options: string);
```

CHAIN adds an entry to the system's program execution queue. When a program terminates, the system examines this queue to see if it contains an entry. If it does the entry is processed as if it were a keyboard response to the X(ecute Command: prompt (i.e., a program and/or redirection options are executed.)

In Version IV, execution options include those that change the prefix volume, change the name of the library text file, and redirect system or program input or output. Consult your Users' Manual for details on these execution options.

Regardless of where a CHAIN call appears in a program, it does not take effect until the program calling it completes execution. If there are a number of calls to CHAIN in a program (or in a program CHAIN'ed to) the chain requests are processed in the order queued.

CHAIN is a standard intrinsic in Version IV. It is available in a number of other implementations as well. In Version IV, programs that use the CHAIN intrinsic must also use the operating system COMMANDIO unit. The unit must be available as a separate file during compilation (for its interface section) and during execution, as a library file or embedded in the operating system. COMMANDIO.CODE is supplied as a standard part of the Version IV distribution.

```
program chtst;
uses commandio;
var myoptions,
     mydisk      : string;
begin
   mydisk := '#5:';  { or any volume name }
   myoptions :=
     concat('*system.filer. PI="K', mydisk, ',YQ"');
   chain(myoptions);
   writeln('After this program terminates it will K(runch ',
      mydisk, '.');
end.
```

This program queues a request to execute the Filer with input that will invoke the Filer K(runch command. The first execution option is `*system.filer.` (The leading asterisk directs the operating system to search the boot volume for `system.filer`; the trailing period prevents the system from attaching a .CODE suffix to the file name before searching for the program.) The `PI` execution option redirects program input from the keyboard to a file or – as in this example – a literal string enclosed in quotes (PI stands for Program Input). The string (`Kmydisk,YQ`) is the text of what a user would type when using the Filer to invoke the K(runch command. A comma is included in the string where the user would ordinarily type a carriage return.

WARNING: In Version IV systems prior to Version IV.13 the CHAIN intrinsic causes the system to temporarily lose the memory space formerly occupied by the program global variables of the calling program. This may result in a stack overflow.

See section 6.14 for further discussion of the CHAIN intrinsic.

4.5 CLOSE

Syntax:

```
procedure close(var F : fileid [; OPTION]);
```

CLOSE sets the file state of the file variable `F` to "closed". `OPTION`s include `LOCK`, `NORMAL`, `PURGE` and `CRUNCH`; these determine the final state of the associated external file. (All options except `PURGE` are ignored when the externalfile is not a disk file.)

`NORMAL` preserves modifications to files opened with RESET; however, if the file was extended, the extension is deleted. If the file was modified, the file date attribute is assigned the current system date. Temporary files created with REWRITE are deleted. `NORMAL` is the default option.

`LOCK` preserves files opened with RESET. If the file was extended, the extension is saved. If the file was modified, the file date attribute is assigned the current system date. Temporary files created with REWRITE become permanent files. Note that if the temporary file's name matches an existing file's name, the existing file is deleted when the temporary file becomes permanent.

`PURGE` deletes the specified disk file. If the external file is an entire volume, the volume is taken off line. This is a meaningless operation for disk volumes, which are automatically remounted by the system. However, deleted serial volumes cannot be opened for subsequent I/O operations until the system is reinitialized.

`CRUNCH` is equivalent to LOCK except that the file is truncated by designating the file window position as the end of the file. (The position of the file window is determined by the last file operation.) All data between the file window and the original end of the file is deleted.

When a structured file is closed, the contents of the file window become undefined. Closing a non-open file causes an I/O error. At procedure termination, a `NORMAL` close is performed on all open files declared in that procedure.

See section 3.3.1 for more information.

4.6 CONCAT

Syntax:

```
function concat(S : string
                {; S : string}) : string;
```

CONCAT returns a string containing the concatentation of the string values of its arguments. Note that CONCAT accepts one or more string parameters.

NOTE: The length of the string result is not allowed to exceed the maximum string size of 255 characters.

NOTE: The INSERT intrinsic (discussed in section 4.15) can be used to accomplish the same effect as CONCAT and is often more efficient.

See section 3.4 for more information on CONCAT.

4.7 COPY

Syntax:

```
function copy(SOURCE : string;
              INDEX  : integer;
              SIZE   : integer) : string;
```

COPY returns a string containing `SIZE` characters copied from `SOURCE`, starting at the `INDEX`'th character position in `SOURCE`.

NOTE: COPY returns an empty string if there are less than `SIZE` characters available from the `INDEX`'th to the last character in `SOURCE`.

See section 3.4 for more information.

4.8 DELETE

Syntax:

```
procedure delete(var S     : string;
                     INDEX : integer;
                     SIZE  : integer);
```

DELETE removes `SIZE` characters from the string in `S`, starting at the `INDEX`'th character position in `S`.

NOTE: DELETE leaves `S` unaltered if there are less than `SIZE` characters available from the `INDEX`'th to the last character in `S`.

See section 3.4 for more information.

4.9 EXCEPTION

Syntax:

```
procedure exception(ChainNoMore: boolean);
```

As mentioned in section 4.4, Version IV of the p-System permits redirection of system and user input and output. The EXCEPTION intrinsic returns the system to its original pre-redirection state. It cancels redirection specified at the outer operating system level (with the X(ecute command) and redirection specified from within a program (via CHAIN or REDIRECT).

If the `chainnomore` boolean is true, the system's execution option queue (filled by the CHAIN intrinsic) is cleared. If the parameter is FALSE the pending CHAINs are unaffected. The primary use of EXCEPTION is to cancel a pre-programmed sequence of inputs and/or program calls because of some exceptional condition.

NOTE: All runtime errors cause the equivalent of a call to EXCEPTION with parameter TRUE.

EXCEPTION is a standard intrinsic in Version IV. It is available in a number of other implementations as well. In Version IV, programs that use the EXCEPTION intrinsic must also use the operating system COMMANDIO unit. The unit must be available as a separate file during compilation (for its interface section) and during execution, as a library file or embedded in the operating system. COMMANDIO.CODE is supplied as a standard part of the Version IV distribution.

4.10 EXIT

Syntax:

```
procedure exit(<routine>);
```

EXIT causes program execution to continue at the end of the block associated with <routine>. Acceptable arguments are procedure or function identifiers, program names, or the reserved word PROGRAM. If EXIT specifies a recursively invoked routine, only the most recent invocation is terminated.

NOTE: `EXIT(PROGRAM)` is the only legal form of the EXIT intrinsic in the initialization or termination sections of a unit. In the initialization section it causes control to pass directly to the termination section of that unit and no further units are initialized, nor is the program executed. In the termination section it aborts execution of the termination section and passes control to the termination section of any unterminated unit. EXIT with a unit identifier is not permitted.

NOTE: Exiting an uncalled procedure results in an execution error ("Exitfrom uncalled procedure").

WARNING: When EXIT is used to terminate a function, the function result must have been assigned beforehand; otherwise, the function returns an undefined value when exited.

See section 3.11.8 for more information.

4.11 FILLCHAR

Syntax:

```
procedure fillchar(var BUFFER : univ;
                       COUNT  : integer;
                       CH     : character);
```

FILLCHAR initializes `COUNT` bytes in memory with the value in `CH`. The starting address is specified by `BUFFER`.

NOTE: The SIZEOF intrinsic may be used as the `COUNT` argument to FILLCHAR; this is useful when an entire data structure must be filled with `CH`.

NOTE: Negative values in `COUNT` are treated as zero byte counts; hence, large unsigned values may not work as expected.

WARNING: FILLCHAR offers no range or type checking.

WARNING: Array indices are treated as signed integers. In the specification of the starting buffer address, use of an array index whose value is less than the buffer's declared lower bound may yield unexpected or fatal results.

See section 3.8 for more information.

4.12 GOTOXY

Syntax:

```
procedure gotoxy(X : integer;
                 Y : integer);
```

GOTOXY moves the cursor to the position specified by its arguments. `X` determines the horizontal displacement, while `Y` determines the vertical displacement; the upper left corner of the screen is defined to be (0,0). If parameter values exceed the maximum values defined for the system terminal they may be truncated to the maximum values.

NOTE: In many programming systems, direct cursor addressing is accomplished by providing a line number followed by a column number. The GOTOXY procedure expects the column number first, followed by the line number.

NOTE: GOTOXY is a terminal-dependent procedure; it usually requires redefinition when a new terminal is incorporated into the system. See your Installation Guide for details on redefining GOTOXY.

See section 3.11.4 for more information.

4.13 HALT

Syntax:

```
procedure halt;
```

HALT suspends the program execution and prints an execution error message indicating that HALT was encountered.

NOTE: Programs may be terminated in a more normal fashion with the EXIT intrinsic. HALT is intended to be used with the system Debugger.

See section 3.11.11 for more information.

4.14 IDSEARCH

Syntax:

```
procedure idsearch(var INX     : integer;
                   var BUFFER  : univ);
```

The IDSEARCH intrinsic is a special-purpose procedure designed to speed up the the search for valid Pascal identifiers in a Pascal source program. It was designed for use in the UCSD Pascal compiler, but it can be useful in any program that accepts Pascal source textas input, such as cross-reference generators, pre-processors and the like.

A detailed exampleusing IDSEARCH appears in section 3.11.13.

4.15 INSERT

Syntax:

```
procedure insert(SUBSTRING : string;
             var S         : string;
                 INDEX     : integer);
```

INSERT stuffs the string in `SUBSTRING` into the string contained in `S` at the `INDEX`'th character position in `S`.

NOTE: When `INDEX` is one more than the length of `S` the effect of INSERT is to concatenate `SUBSTRING` to the end of `S`. If `INDEX` is any greater than this value, INSERT leaves `S` unaltered.

See section 3.4 for more information.

4.16 IORESULT

Syntax:

```
function ioresult : integer;
```

IORESULT returns an integer value indicating the result of the last I/O operation performed by the current task (section 3.0).

NOTE: I/O results are updated after every I/O operation; therefore, an I/O result value must be saved in a variable if subsequent I/O operations occur before it can be manipulated.

NOTE: A table of standard I/O error numbers and their corresponding messages is contained in Appendix A. Conditions causing bad I/O results are listed in Appendix C.

See section 3.11.9 for more information.

4.17 LENGTH

Syntax:

```
function length(S : string) : integer;
```

LENGTH returns the dynamic length of the string contained in S.

See section 3.4 for more information.

4.18 MARK

Syntax:

```
procedure mark(var MARKP : ↑integer);
```

MARK opens a "sub"-heap for dynamically allocated variables. Subsequent calls to NEW allocate dynamic variables only in the new heap. The heap is identified by the value assigned to MARKP. The RELEASE intrinsic is used to deallocate all dynamic variables in a heap opened by MARK.

NOTE: New heaps are allocated within the current heap; thus, heaps are nested. Deallocating a given heap results in the deallocation of all subsequently opened heaps.

MARK causes the system to record the address of the current top-of-heap in MARKP. When RELEASE is invoked with the same pointer variable, the top-of-heap is set back to that location.

WARNING: Pointers passed to MARK must only be used as arguments to subsequent calls to RELEASE. Careless use of MARK and RELEASE leads to "dangling references" (i.e., pointers to deallocated dynamic variables, which may or may not be overwritten by subsequent system actions).

See section 3.5 for more information.

4.19 MEMAVAIL

Syntax:

```
function memavail : integer;
```

MEMAVAIL returns the number of unused words in memory. The integer result contains an unsigned value; if large, it may be misinterpreted as a negative value unless specifically treated as an unsigned integer result (see section 6.2).

NOTE: The MEMAVAIL intrinsic (section 3.5.1) returns the number of words between the system stack and heap. It should be used on Version II systems. The VARAVAIL intrinsic (section 3.5.2) is a more useful intrinsic for sizing memory in Version IV. In Version IV, MEMAVAIL returns the number of words between the stack and the heap exclusive of any internal codepool (which may "float" in the area between the stack and the heap). This is not a true reflection of available memory space under Version IV because there may be unneeded segments that are in memory or required segments that are not in memory. MEMAVAIL does not consider space occupied by unneeded segments as available space. The MEMAVAIL and VARAVAIL intrinsics return equivalent values on implementations using an external code pool.

See section 3.11.10 for more information.

4.20 MEMLOCK

Syntax:

```
procedure memlock(SEGLIST : string);
```

MEMLOCK locks into memory the code of each segment named in the SEGLIST. Non-resident segments named in SEGLIST are read into memory and locked. A MEMLOCKed segment remains in memory until it is named in a call to MEMSWAP (section 4.21) and the operating system needs the space it occupies.

The SEGLIST consists of a list of segment names separated by commas; spaces are ignored. It may contain any segment name declared either in the program and the units it uses, or in the operating system. Unrecognized segment names are ignored.

WARNING: On systems using an internal code pool, indiscreet calls to MEMLOCK may render the heap incapable of providing large continuous blocks of memory.

See section 3.2 for more information.

4.21 MEMSWAP

Syntax:

```
procedure memswap(SEGLIST : string);
```

MEMSWAP causes the code of each MEMLOCKed segment (see section 4.20) named in the `SEGLIST` to be flagged as unloadable. The `SEGLIST` consists of a list of segment names separated by commas; spaces are ignored. It may contain any segment name declared either in the program and its used units, or in the operating system. Unrecognized segment names are ignored.

NOTE: MEMSWAP operates only on MEMLOCKed segments. A MEMLOCKed code segment is not unloaded until a MEMSWAP call has been performed for each MEMLOCK call naming that segment, the segment is not executing and the memory space it occupies is required by the operating system.

See section 3.2 for more information.

4.22 MOVELEFT

Syntax:

```
procedure moveleft(var SOURCE      : univ;
                       DESTINATION : univ;
                       COUNT       : integer);
```

MOVELEFT moves `COUNT` bytes of data from the buffer addressed by `SOURCE` to the buffer addressed by `DESTINATION`. The data is moved one byte at a time, starting with the bytes addressed by `SOURCE` and `DESTINATION`, and moving successively higher-addressed bytes until `COUNT` bytes have been moved.

WARNING: Negative values in `COUNT` are treated as zero byte counts; hence, large unsigned values may not work as expected.

WARNING: MOVELEFT does not perform type or range checking on its parameters.

WARNING: Array indices are treated as signed integers. In the specification of the starting buffer address, use of an array index whose value is less than the buffer's declared lower bound may yield unexpected or fatal results.

WARNING: MOVELEFT should not be used to move transfer data between strings and packed arrays of char. A string is actually one byte longer than its declared size; the leading byte of a string contains the length of the string.

See section 3.8 for more information.

4.23 MOVERIGHT

Syntax:

```
procedure moveright(var SOURCE      : univ;
                        DESTINATION : univ;
                        COUNT       : integer);
```

MOVERIGHT moves `COUNT` bytes of data from the buffer addressed by `SOURCE` to the buffer addressed by `DESTINATION`. The data is moved one byte at a time, starting with the bytes addressed by the expressions `(SOURCE + COUNT - 1)` and `(DESTINATION + COUNT - 1)`, and moving successively lower-addressed bytes until `COUNT` bytes have been moved.

WARNING: Negative values in `COUNT` are treated as zero byte counts; hence, large unsigned values may not work as expected.

WARNING: MOVERIGHT does not perform type or range checking on its parameters.

WARNING: Array indices are treated as signed integers on most p-System implementations. In the specification of the starting buffer address, use of an array index whose value is less than the buffer's declared lower bound may yield unexpected or fatal results.

WARNING: MOVERIGHT should not be used to move transfer data between strings and packed arrays of char. A string is actually one byte longer than its declared size; the leading byte of a string contains the length of the string.

See section 3.8 for more information.

4.24 OPENNEW

Syntax:

```
procedure opennew(var F        : fileid;
                      FILENAME : string);
```

OPENNEW is equivalent to the REWRITE intrinsic. Section 4.32 describes REWRITE.

NOTE: OPENNEW is not recognized in Version IV of the p-System.

4.25 OPENOLD

Syntax:

```
procedure openold(var F        : fileid
                  [; FILENAME : string]);
```

OPENOLD is equivalent to the RESET intrinsic. Section 4.31 describes RESET.

NOTE: OPENOLD is not recognized in Version IV of the p-System.

4.26 P_MACHINE

Syntax:

```
procedure pmachine(<item> {,<item>});
```

PMACHINE generates in-line machine code corresponding to the items specified in the parameter list. See your Internal Architecture Guide for a description of the p-machine instruction set.

```
<item> ::=  <code>               |
            <expression>         |
            <address-reference>

<code>    ::=  A constant value or constant identifier
               denoting an integer between 0 and 255;
               PMACHINE emits a single byte with the
               specified value.  Values outside this
               range have only their least significant
               byte emitted.  Code bytes represent
```

```
                    either p-code instructions or instruction
                    operands.

<expression> ::= (<Pascal expression>)

<Pascal expression> ::=
                         An expression (e.g., the right-hand
                         side of an assignment statement).
                         PMACHINE generates code which
                         evaluates the expression, leaving
                         the result on the stack.

<address-reference> ::=  ↑<variable reference>

<variable reference> ::=
                          A referenced variable (e.g., the
                          left-hand side of an assignment
                          statement).  PMACHINE generates
                          code which evaluates the variable
                          reference, leaving the address on
                          the stack.
```

See section 3.10 for more information.

4.27 POS

Syntax:

```
function pos(SUBSTRING : string;
             S         : string) : integer;
```

POS searches S for an occurrence of SUBSTRING. It returns an index indicating the start of the matched substring. If S contains multiple occurrences of SUBSTRING, POS locates the first occurrence. If S contains no occurrences of SUBSTRING, POS returns 0.

See section 3.4 for more information.

4.28 PWROFTEN

Syntax:

```
function pwroften(EXPONENT : integer) : real;
```

PWROFTEN returns the floating point representation of ten raised to the EXPONENTth power. The maximum value of EXPONENT is system dependent; consult your documentation. Negative arguments to PWROFTEN and arguments greater than the maximum exponent cause an execution error ("Floating point error ").

See section 3.11.6 for more information.

4.29 REDIRECT

Syntax:

```
function redirect(EXECUTION_OPTIONS: string): boolean;
```

REDIRECT causes I/O redirection to take place as specified in the `EXECUTION_OPTIONS` string. If the redirection is successfully accomplished, REDIRECT returns TRUE; otherwise, it returns FALSE. The `EXECUTION_OPTIONS` string may contain prefix, library, input and output options, but not a program name. (The CHAIN intrinsic may be used to initiate execution of another program.)

Consult your System Users' Manual for specific execution options. An example using a number of execution options can be found in section 4.4 of this text.

WARNING: If REDIRECT returns FALSE, the state of I/O redirection is undefined and programs may not function as intended. The system can be returned to its original state of redirection using the EXCEPTION intrinsic (see section 4.9).

WARNING: The REDIRECT intrinsic causes the system to temporarily lose space in memory. Memory formerly occupied by heap variables prior to a REDIRECT may not be available after the REDIRECT. This may result in a stack overflow.

REDIRECT is a standard intrinsic in Version IV of the p-System. It is available in a number of other implementations as well. REDIRECT uses the operating system COMMANDIO unit. The unit must be available as a separate file during compilation (for its interface section) and as a library file during execution. COMMANDIO.CODE is supplied as a standard part of the Version IV distribution.

4.30 RELEASE

Syntax:

```
procedure release(var MARKP : ↑integer);
```

RELEASE deallocates all dynamic variables in the heap associated with `MARKP`. It takes the value of the `MARKP` pointer, which indicates a prior top-of-heap, and makes it the current top-of-heap.

NOTE: New heaps are allocated within the current heap; thus, heaps are nested. Deallocating a given heap results in the deallocation of all subsequently opened heaps.

NOTE: RELEASEing the current heap does not affect the memory occupied by code segments.

WARNING: Pointers passed to RELEASE must be initialized by a previous call to the MARK intrinsic. Careless use of MARK and RELEASE leads to "dangling references" (i.e., pointers to deallocated dynamic variables, which may or may not be overwritten by subsequent system actions).

WARNING: RELEASE deallocates dynamic variables allocated by user programs. It does not deallocate variables allocated by the system through calls to the CHAIN and REDIRECT intrinsics.

WARNING: RELEASE also deallocates task stacks allocated by calls to the START intrinsic after a call to the MARK intrinsic. This may cause a system crash.

See section 3.5 for more information.

4.31 RESET

Syntax:

```
procedure reset(var F          : fileid
                  [; FILENAME : string]);
```

RESET opens the existing external file named in `FILENAME` for reading and/or writing, and prepares the file variable `F` for subsequent operations on the external file. If `F` does not denote an interactive file, RESET performs an implicit GET. This is consistent with the standard procedure RESET in Standard Pascal.

RESET generates an I/O error in the following cases:

- File variable `F` is already open.
- `FILENAME` specifies a nonexistent external file.

RESET without the file name parameter rewinds the file window to the beginning of the (open) file.

External files opened with RESET may be closed with the CLOSE intrinsic.

See section 3.3 for more information.

4.32 REWRITE

Syntax:

```
procedure rewrite(var F        : fileid;
                      FILENAME : string);
```

REWRITE creates a temporary external file named `FILENAME`, and prepares the file variable `F` for subsequent operations on the external file.

REWRITE is used to open new files for writing.

REWRITE generates an I/O error in the following cases:

- The file variable `F` is already open.
- Insufficient room on disk to create the file.

External files opened with REWRITE may be saved (and optionally made permanent) with the CLOSE intrinsic.

See section 3.3 for more information.

4.33 SCAN

Syntax:

```
function scan(COUNT : integer;
              <partial expression>;
          var BUFF  : univ) : integer;
```

Starting at the address specified by the variable `BUFF`, SCAN examines successive bytes in memory until one of the following conditions becomes true:

- The current byte contains a value which satisfies the partial expression.
- `COUNT` bytes have been examined without finding a value that satisfies the partial expression.

Partial expressions are incomplete "equal-to" or "not-equal-to" Boolean expressions with a character expression as the right-hand operand (e.g.,= ';'). The left-hand operand is missing from a partial expression; it is defined to be the current byte being examined by SCAN. A partial expression is satisfied when it evaluates to true. The partial expression is evaluated for each byte examined by SCAN; if it becomes true, SCAN returns immediately.

```
<partial expression> ::=  = | <>   <character expression>

<character expression> ::= character variable or constant
```

SCAN returns the number of bytes examined. If the byte pointed to by the starting address contains a value satisfying the partial expression, SCAN returns 0. If the value in COUNT is negative, SCAN scans backwards (towards lower addresses) from its starting address searching for the target byte, and returns a negative number (in the range COUNT..0) whose magnitude indicates the number of bytes examined.

WARNING: Negative values in COUNT cause backwards scanning; hence, large unsigned values may not work as expected.

WARNING: Array indices are treated as signed integers. In the specification of the starting buffer address, use of an array index whose value is less than the buffer's declared lower bound may yield unexpected results.

See section 3.8 for more information.

4.34 SEEK

Syntax:

```
procedure seek(var F      : fileid;
                   RECNUM : integer);
```

SEEK moves the file window in F so that a subsequent GET or PUT accesses the RECNUMth record in the file. F must be a structured disk file (i.e., any Pascal file except text). The first record in a file is record 0. The standard procedure EOF is used to detect seeks off the end of the file. Though SEEK itself always sets EOF to false, a subsequent GET sets EOF to true if the new file position is at (or past) the end of the file.

WARNING: The result of SEEK is undefined if the file position is moved more than one record past the final record in the file. If SEEK moves to the first empty record past the end of the file, a subsequent PUT extends the file in a normal fashion; however, if records are written at file positions more than one record past the end of the file, the file itself becomes undefined (resulting in subsequent program errors). Furthermore, other data files may be overwritten. Note that EOF alone is insufficient to distinguish these cases; the programmer must keep track of the current last record in the file explicitly.

See section 3.3.5 for more information.

4.35 SEMINIT

Syntax:

```
procedure seminit(var SEM   : semaphore;
                      COUNT : integer);
```

SEMINIT initializes `SEM` with the value `COUNT`.

WARNING: Calling SIGNAL or WAIT with an uninitialized semaphore variable may crash the system. Calling SEMINIT with a semaphore holding suspended tasks causes the system to lose the tasks.

See section 3.0.1 for more information.

4.36 SIGNAL

Syntax:

```
procedure signal(var SEM : semaphore);
```

If no tasks are waiting on `SEM`, `SIGNAL` increments the semaphore's count; otherwise, SIGNAL selects the highest priority waiting task and inserts it in the ready queue.

See section 3.0.1 for more information.

4.37 SIZEOF

Syntax:

```
function sizeof(<identifier>) : integer;
```

SIZEOF returns the number of bytes of memory allocated for the data object denoted by <identifier>. <identifier> may be either a variable or type identifier. SIZEOF is used in conjunction with the intrinsics MOVELEFT, MOVERIGHT and FILLCHAR.

NOTE: SIZEOF is evaluated by the compiler; it replaces each call with a constant containing the result. Thus, SIZEOF cannot return the size of runtime variable references.

NOTE: If a record contains variant fields, SIZEOF uses the longest variant when determining its size.

NOTE: SIZEOF does not work with individual array elements or with explicitly qualified fields of records. (It does work with fields of records when qualified using WITH.)

See section 3.8 for more information.

4.38 START

Syntax:

```
procedure start(<process call>
          [; var PID       : processid
             [; STACKSIZE : integer
             [; PRIORITY  : integer ]]]);
```

START initiates tasks.

The main parameter to START is a process call; it resembles a procedure call, and may contain parameters passed to the task.

The remaining parameters define various task attributes. `PID` is assigned a value which uniquely identifies the new task. `STACKSIZE` indicates the number of words of memory to be allocated for a stack space; if absent, START uses 200 as a default stack size. `PRIORITY` indicates the task priority to be assigned the new task; the higher the number, the higher the priority assigned to that process. if the priority is not in the range 0..255, an execution error occurs. The default priority is 128.

See section 3.0 for more information.

4.39 STR

Syntax:

```
procedure str(L : integer[36];
          var S : string);
```

STR converts the value in `L` into a string in `S`; it is used to format long integer values for output. If the value in `L` is negative, the first character placed in `S` is a minus sign (-).

See section 3.6 for more information.

4.40 TIME

Syntax:

```
procedure time(var HIWORD : integer;
               var LOWORD : integer);
```

TIME returns the current value of the system clock in the integer pair HIWORD and LOWORD. The system clock is an unsigned 32-bit integer incremented every 60th of a second. HIWORD contains the most significant word.

NOTE: HIWORD and LOWORD contain unsigned values; they may be treated as negative numbers by some integer operations unless specifically treated as unsigned integers (see section 6.2).

NOTE: TIME returns the time relative to the time of the system bootstrap, not the true time of day. If a machine does not support a clock (or the clock is turned off) TIME generally returns zero in both HIWORD and LOWORD.

See section 3.11.5 for more information.

4.41 TREESEARCH

Syntax:

```
type alpha = packed array [1..8] of char;
     nodeptr = ↑node;
     node = record
              name: alpha;
              right_link: nodeptr;
              left_link: nodeptr;
              { any user-defined record fields }
            end {node};

function treesearch(ROOT : nodeptr;
                var NODE : nodeptr;
                    NAME : alpha) : integer;
```

TREESEARCH manages binary trees ordered by the contents of an 8-character field. TREESEARCH searches the tree rooted at ROOT for a record whose name field matches NAME. On return, NODE contains a pointer to the last record examined, and the function result indicates the result of the search.

If a record in the tree matches the array argument, TREESEARCH returns 0 as a function result and NODE points to the matching record. If no record in the tree matches the array argument, TREESEARCH returns either 1 or -1 and NODE is set to the last node searched. 1 indicates that NAME is greater than the name in the record pointed at by NODE (and would be inserted on its right link); -1 indicates that the argument is less than NODE (and would be inserted on its left link).

NOTE: The TREESEARCH intrinsic constructs trees so that either right (post-order) traversals or left (pre-order) traversals visit the records in lexicographical order of their name fields. The particular ordering used is implementation dependent.

See section 3.11.12 for more information.

4.42 UNITBUSY

Syntax:

```
function unitbusy(UNITNUM : integer) : boolean;
```

UNITBUSY indicates whether the specified device is waiting for an I/O operation to finish.

UNITBUSY is not available in most p-System implementations.

See section 3.9 and Appendix D for more information.

4.43 UNITCLEAR

Syntax:

```
procedure unitclear(UNITNUM : integer);
```

UNITCLEAR cancels any I/O operations occurring on the specified device, and resets the unit to its initial (i.e., power-up) state.

See section 3.9 and Appendix D for more information.

4.44 UNITREAD

Syntax:

```
procedure unitread(UNITNUM  : integer;
               var BUFF     : univ;
                   COUNT    : integer
                [; BLOCKNUM : integer
                [; CONTROL  : integer]]);
```

UNITREAD reads `COUNT` bytes from the device `UNITNUM` into the variable `BUFF`. `BLOCKNUM` is applicable only when reading from block-structured units; it specifies the starting block of the transfer. (Block numbers start at 0.) `CONTROL` is treated as a bit array; certain bits in the control word are defined to select various I/O options (depending on the unit specified –see Appendix D for details).

The `BLOCKNUM` parameter is ignored when `UNITNUM` specifies a serial unit. Though it is not specified in the syntax definition above, UNITREAD accepts a `CONTROL` parameter in the absence of a `BLOCKNUM` parameter. The form is:

```
UNITREAD(<unit>,<buffer>,<length>,,<control>)
```

NOTE: On most implementations, `BUFF` is constrained to start on a word address when the specified unit is block-structured. Reading into an odd byte address causes an I/O error (illegal buffer address).

WARNING: UNITREAD performs no type or range checks on its parameters.

WARNING: Array indices are treated as signed integers. In the specification of the starting buffer address, use of an array index whose value is less than the buffer's declared lower bound may yield unexpected or fatal results.

See section 3.9 and Appendix D for more information.

4.45 UNITSTATUS

Syntax:

```
procedure unitstatus(UNITNUM   : integer;
                 var STATREC   : univ;
                     DIRECTION : integer);
```

UNITSTATUS returns the status of the device `UNITNUM` in the `STATREC` record. The format of the `STATREC` record depends on the type of device being polled and on the particular p-System Version/hardware environment. It is normally a serial device record or a block-structured device record. The record should occupy at least 30 words to allow for future expansion.

The `DIRECTION` parameter should be passed as zero or one. For devices that perform both input and output, zero specifies the status information referring to output and one specifies the status information referring to input.

The DIRECTION parameter is not used under all implementations or for all devices.

See section 3.9 and Appendix D for more information.

4.46 UNITWAIT

Syntax:

```
procedure unitwait(UNITNUM : integer);
```

UNITWAIT waits for the device specified by UNITNUM to finish its current I/O operation.

NOTE: UNITWAIT is not available in most p-System implementations.

See section 3.9 and Appendix D for more information.

4.47 UNITWRITE

Syntax:

```
procedure unitwrite(UNITNUM  : integer;
                var BUFF     : univ;
                    COUNT    : integer
                  [; BLOCKNUM : integer
                  [; CONTROL  : integer]]);
```

UNITWRITE writes COUNT bytes to the device UNITNUM from the variable BUFF. BLOCKNUM is applicable only when writing to block-structured units; it specifies the starting block for the transfer. (Block numbers start at 0.) CONTROL is treated as a bit array; certain bits in the control word are defined to select various I/O options (depending on the unit specified –see Appendix D for details).

The BLOCKNUM parameter is ignored when UNITNUM specifies a serial unit. Though it is not specified in the syntax definition above, UNITWRITE accepts a CONTROL parameter in the absence of a BLOCKNUM parameter. The form is:

```
UNITWRITE(<unit>,<buffer>,<length>,,<control>)
```

NOTE: On most implementations, `BUFF` is constrained to start on a word address when the specified unit is block-structured. Writing from an odd byte address causes an I/O error (illegal buffer address).

WARNING: UNITWRITE performs no type or range checks on its parameters.

WARNING: Array indices are treated as signed integers. In the specification of the starting buffer address, use of an array index whose value is less than the buffer's declared lower bound may yield unexpected or fatal results.

See section 3.9 and Appendix D for more information.

4.48 VARAVAIL

Syntax:

```
function varavail(SEGLIST : string) : integer;
```

The VARAVAIL function returns the size, in words, of the free space in memory assuming that all segments named in the `SEGLIST` are resident. The `SEGLIST` consists of a list of segment names separated by commas; spaces are ignored. It may contain any segment name declared either in the program and the units it uses, or in the operating system. Unrecognized segment names are ignored.

VARAVAIL returns the same value as MEMAVAIL on systems with an external code pool.

See section 3.5 for more information.

4.49 VARDISPOSE

Syntax:

```
procedure vardispose(var P         : ↑univ;
                         WORDCOUNT : integer);
```

The VARDISPOSE procedure deallocates the `WORDCOUNT`-sized data structure referenced by pointer `P`. `WORDCOUNT` is an unsigned integer parameter. `P` is returned containing NIL.

WARNING: Deallocating a data structure of a different size than was originally allocated could lead to a system crash.

NOTE: Attempts to deallocate one word actually deallocate two words.

See sections 3.5 and 4.50 for more information.

4.50 VARNEW

Syntax:

```
function varnew(var P         : ↑univ;
                    WORDCOUNT : integer) : integer;
```

The VARNEW function attempts to allocate a data structure of WORDCOUNT words on the heap and return P as a pointer to the data structure. P has been declared as a pointer to the type of the data structure being allocated. If there is enough contiguous free memory for the buffer, the allocation takes place and the value of VARNEW is returned equal to WORDCOUNT; otherwise, VARNEW is returned zero.

The SIZEOF intrinsic may be applied to the data structure being allocated to determine the value of WORDCOUNT. The system does not check to assure that there is a correspondence between the size of the data structure pointed to by P and the value of WORDCOUNT.

NOTE: Attempts to allocate one word actually allocate two words.

See section 3.5 for more information.

4.51 WAIT

Syntax:

```
procedure wait(var SEM : semaphore);
```

If the semaphore count of SEM is greater than zero, it is decremented, and the current task continues to execute. Otherwise, the current task is suspended, and waits for a SIGNAL on SEM.

See section 3.0.1 for more information.

Chapter 5

COMPILE OPTIONS

Contents

5.0 Options 142
5.0.1 Compiled Listings 145
5.0.2 Include Files 147
5.0.3 Using Units 148
5.0.4 Swapping Compiler 149
5.0.5 Conditional Compilation 150
5.0.6 I/O Checks 151
5.0.7 Range Checks 152
5.0.8 GOTO Restriction 153
5.0.9 Copyright Notices 153
5.0.10 Console Display Suppression 154
5.0.11 Segment Residency 154
5.0.12 System Programs 155
5.0.12.1 System Programs Before Version IV 156
5.0.12.2 System Programs Under Version IV 159
5.0.13 Native Code 159
5.0.14 Real Size 160
5.0.15 Symbolic Debugging 161
5.0.16 Byte Sex Flipping 161
5.0.17 Executable Units 162
5.0.18 "Tiny" Compiler 165

5.1 Option Summary 165

This chapter describes the compile options in UCSD Pascal. Compile options affect both compiler operation and the execution characteristics of code produced by the compiler. Compile options are controlled by directives embedded in the text of source programs. They are processed by the compiler as they are encountered in the program. Section 5.0 describes the use of compile options, and provides a detailed description of each compile option. Section 5.1 summarizes the compile options.

5.0 Options

Compile options appear as directives in a source program. These directives are called **pseudo-comments**. A pseudo-comment is a comment (as defined in UCSD Pascal) which contains a ”$” character immediately following the left-hand comment delimiter. Following the ”$” is a list of one or more compile options; multiple options are delimited by commas. Each compile option consists of a single alphabetic character (upper or lower case) denoting a specific option, possibly followed by an argument.

An option which may accept ”+”, ”-” or ”^” as arguments is known as a **switch option**. Compile options which accept alpha-numeric arguments are known as **string options**. These arguments may be integers, lists of UCSD Pascal identifiers, file names or simply text strings. String options are terminated by the right-hand comment delimiter. Note that a single pseudo-comment cannot contain more than one string option, as the comma which normally separates multiple options (and all that follow it until the end of the comment) would be considered part of the original string option.

NOTE: String arguments may not contain the character ”*” when the right-hand comment delimiter is ”*)”, and may not contain the character ”}” when the right-hand delimiter is ”}”.

WARNING: Generally, invalid compile options are ignored by the compiler; the pseudo-comment is treated as a normal comment. One exception to this is when a string option contains an argument violating the restriction described in the NOTE above. In that case the string argument is erroneously truncated. If the illegal character is the first character in the string, and the option character happens to be used for both string and switch options, the string option is incorrectly treated as a switch option –beware!

Examples of compile options:

```
{$I+}
(*$I yeenly.text *)
{$L+,U-,S+}
(*$L+,U- *)
(*$B CondIdent- *)
{$P}
```

Example of string option incorrectly treated as switch option:

```
(*$I*dysfunc.text *)
```

All switch options accept the "+" and "-" switches as arguments. The "+" switch enables the option (on) while the "-" switch disables the option (off). Values for the D, I, N and R options (and the L option under Version IV.13) may be stacked up to 15 levels deep. Thus, when a directive is set (e.g., R- or I+), the new value is pushed onto the top of its stack. An option's current value is the value on the top of its stack. The "^" switch pops the option's stack, causing the option to be restored to its prior value. Stacked options are useful when a short section of code requires the assertion of an option value, but the option value of the enclosing program is unknown or subject to change. A directive pair of the form: {$I-} ... {$I^} asserts the desired compile option value without affecting the option value in the enclosing program.

Attempts to stack an option to a depth greater than 15 cause the values on the bottom of the stack to be lost.

Example of stacked compile option values:

```
program stack;
var i: integer;
begin
  repeat
    {$I-}
    readln(i);
    {$I↑}
  until ioresult = 0;
  writeln ('value is: ',i);
end {stack}.
```

Syntax for compile options:

```
<pseudo-comment> ::=  <L-delim>$<options><R-delim>

<L-delim> ::=  UCSD Pascal comment delimiter: "{" or "(*"

<R-delim> ::=  UCSD Pascal comment delimiter: "}" or "*)"

<options> ::=  <string-option>  |  <option-list>
```

```
<option-list> ::=  <switch-option-list>[,<string-option>]

<switch-option-list> ::=  <switch-option>{,<switch-option>}

<switch-option> ::=  <switch-directive><switch> |
                     <button-directive>

<string-option> ::=  <string-directive><string>  |
                     <idlist-directive><idlist>  |
                     <cond-directive><flag>[<switch>]

<string> ::=  any sequence of characters other than
              "*" or "}" (see previous NOTE)

<idlist> ::=  <identifier>{,<identifier>}

<flag> ::=  A compile flag identifier which follows the
            same rules as a UCSD Pascal identifier.

<switch> ::=  "+"  |  "-"  |  "↑"

<switch-directive> ::=  I  |  L  |  Q  |  R  |  S  |  U  |
                        G  |  N  |  D  |  F  |  V  |  H  |  T

<button-directive> ::=  P  |  R2 |  R4

<string-directive> ::=  C  |  I  |  L  |  T

<idlist-directive> ::=  R

<cond-directive> ::=  B  |  D  |  E
```

Switch directive	Compile option
I	I/O checking (section 5.0.6)
L	Listing (section 5.0.1)
Q	Console display (section 5.0.10)
R	Range checking (section 5.0.7)
V	String parameter range checking (section 5.0.7)
S	Swapping compiler (section 5.0.4)
U	User lex level (section 5.0.12)
G	GOTO (section 5.0.8)
N	Native code generation (section 5.0.13)
D	Symbolic debugging (section 5.0.15)
F	Byte sex flipping (section 5.0.16)
H	Executable units (section 5.0.17)
T	"Tiny" compiler (section 5.0.18)
N	Nonresident unit (section 5.0.11)

Button directive	Compile option
P	Page eject during listing (section 5.0.1)
R2	Generate code for two-word reals (section 5.0.14)
R4	Generate code for four-word reals (section 5.0.14)

String directive	Compile option
C	Copyright notice (section 5.0.9)
I	Include file (section 5.0.2)
L	Compiled listing (section 5.0.1)
T	Listing title (section 5.0.1)

Id List directive	Compile option
R	Resident segment (section 5.0.11)

Conditional directive	Compile option
B	Beginning of section (section 5.0.5)
D	Identifier declaration (section 5.0.5)
E	End of section (section 5.0.5)

5.0.1 Compiled Listings

Compiled listings serve two purposes in UCSD Pascal. First, they provide a complete listing of the program source in a single text file. This is useful when the program source itself resides in a number of textfiles which are included during compilation. Second (and more important), they serve as a debugging tool; a compiled listing contains information used to locate the Pascal source statement responsible for causing an execution error (see section 6.11 for details).

Example of a compiled listing:

```
Pascal Compiler IV.13 c6t-4      12/ 1/84            Page   1

    1   0     0:d    1  {$L look.text}
    2   2     1:d    1  program example;
    3   2     1:d    1    var i,j,k: integer;
```

```
 4  2   1:d   4        s1,s2: string;
 5  2   1:d  86        r: real;
 6  2   1:d  90
 7  2   1:d  90  segment procedure stuff;
 8  3   1:d   1    var l1,l2: integer;
 9  3   1:d   3
10  3   1:d   3  procedure local;
11  3   2:0   0  begin
12  3   2:1   0    writeln('in stuff');
13  3   2:1  20    exit(program);
14  3   1:0   0  end;
15  3   1:0   0
16  3   1:0   0  begin
17  3   1:1   0    if i = 45 then local;
18  3   1:0   0  end {stuff};
19  3   1:0   0
20  2   1:d   1  segment function max(a,b:integer)
21  2   1:d   3                         :integer;
22  5   1:0   0  begin
23  5   1:1   0    if a < b then max := b
24  5   1:1   5             else max := a;
25  5   1:0   0  end {max};
26  5   1:0   0
27  2   1:0   0  begin
28  2   1:1   0     i := 45;
29  2   1:1   4     r := 4.4E1;
30  2   1:1  10     if max(i,trunc(r)) = i
31  2   1:1  18       then i := 45;
32  2   1:1  25     stuff;
33  2    :0   0  end {example}.

End of Compilation.
```

The first column displays the line number (in the listing) of the current source line. The second column displays the segment number of the code segment containing the code corresponding to the source line.

The third column displays two numbers separated by a colon. The left-hand number displays the procedure number of the procedure which contains the code corresponding to the source line. The right-hand number displays the current nesting level of the source statement. The nesting level is determined by the number of unterminated BEGIN/END pairs enclosing the source statement. Note that the nesting level is replaced by the letter "d" when the corresponding source line contains declarations rather than statements.

The fourth column displays the code offset of the corresponding source statement, or the data offset of the corresponding declarations (indicated by the presence of a "d" in the previous column). Code offsets are byte offsets from the beginning of the current procedure. Data offsets are word offsets into the data space of the enclosing block. In both cases, the value displayed represents the offset of the beginning of the code or data item which will be generated by the compiler for the current line.

Compiled listings are generated when the List option is enabled. The List option is controlled by the pseudo-comment directive "L", which is used as both a switch option and a string option.

The default setting of the List option is off. "L+" enables the List option, and produces a compiled listing written to the disk file *SYSTEM.LST.TEXT. The compiled listing may be written to a different file name by using "L" as a string option. This enables the List option and specifies a user-defined list file name.

Portions of a program may be listed by selectively enabling ("L+") and disabling ("L-") the List option.

List files are saved whether or not the compiler flags syntax errors. If errors occur, error messages are embedded in the list file.

NOTE: "L" may be used only once as a string option during the course of a compilation.

Page breaks may be placed in a compiled listing by using the Page option. "P" emits a single page break.

Version IV of the p-System permits user-specified titles to be listed, instead of the default title shown in the example above. The "T" string option may be used at any time in a program to change the current title to the one specified by the string.

Example of listing directives:

```
{$T I want this title printed }
{$L mylist.text }
{$L+}
{$L-}
{$P}
```

5.0.2 Include Files

The include file facility allows the source comprising a large program to be distributed among a number of relatively small and easy-to-manage text files. The compiler accepts only one source file as an input file. However, the input file may contain an include directive for each include file required. When the compiler finds an include directive, it includes the contents of the specified text file as program source. When the end of the include file is reached, the compiler returns to the source following the original include directive.

Include directives may appear anywhere in a source file.

NOTE: Include files may be nested up to ten deep. Certain restrictions on the use of include files arise when compiling units (see section 3.2 for details).

Include files are specified by the pseudo-comment directive "I" used as a string option. The string contains an include file name, which does not require a file suffix. If the file cannot be opened as specified, the compiler appends ".TEXT" to the file name and attempts to reopen it. If this also fails a syntax error occurs.

WARNING: If an error occurs during a compilation, the compiler optionally permits the user to return to the system Editor to correct the problem. The compiler marks the location in the source program where the error occurred, and the Editor jumps to that location upon being invoked. If the error occurs in an included file, the user must be sure to specify the name of that file to the Editor, *not* the name of the host file. If the host file is the system work file, the Editor automatically attempts to use it; the user must prevent the Editor from reading the host (some versions of the Editor permit the user to defeat the default) or save the host so that it is no longer the system work file.

When using the include directive, the compiler must read in volume directories to locate the specified files. This takes up memory space that would otherwise be utilized by the compilation process itself. On pre-Version IV releases of the p-System (including Apple Pascal) it is possible to establish a 4-block file called SYSTEM.SWAPDISK on the system volume. This permits the compiler to swap some of its work space to disk while the directory is read in, and return it to memory when the included file has been located and the directory is no longer needed.

Example of include directive:

```
{$I foon.text}
{$I 3.2:globals }
```

See section 3.11.2 for an example of include files.

5.0.3 Using Units

When a program uses a UNIT the compiler compiles the UNIT's interface section into the host program. The interface section is stored in the code file containing the unit. The compiler looks for the UNIT (if it is not in the same file as the host) in the system library unless directed to look in another library with the "U" string option. The string contains the name of the code file containing the desired unit, including any volume name or file suffix (such as ".CODE"). The specified code file remains the active library through the end of the compilation or until it is overwritten by another $U directive. A $U directive must appear before the USES statement for that unit.

Note that even if a library text file (such as USERLIB.TEXT) names the desired libraries the compiler still requires a "U" directive in order to locate the UNIT.

Example of unit directive:

```
{$U MYLIB.CODE} IOPROCS;
uses {$U main:kaboz.code} kaboozee;
```

See section 3.2 for further information on using UNITs.

5.0.4 Swapping Compiler

In pre-Version IV releases (including Apple Pascal) the compiler may assume an alternate mode of operation for compiling large programs. In these versions, the compiler normally operates as a single memory-resident segment. This mode is used to compile programs that do not tax the system's compile-time memory resources. The swapping option transforms the compiler into two separate disk-resident segments (one for handling statements and the other for handling declarations). This provides extra memory space for the compilation of large programs.

Swapping mode saves about four thousand words of memory during compilation, but halves the compile speed when the compiler code file resides on a diskette.

The swapping option is ignored on Version IV releases of the p-System; the compiler automatically swaps as necessary.

The swapping option is controlled by the pseudo-comment directive "S" used as a switch option. The default setting of the swapping option is off. "S+" enables swapping.

NOTE: "S++" may be used to cause the compiler to do even more swapping. Use of this option provides approximately 1500 additional words of memory.

NOTE: Swapping option directives must appear before the program or unit heading. Unlike other options, swapping cannot be selectively turned on and off during compilation.

Example of swapping directive:

```
{$S+}
```

5.0.5 Conditional Compilation

Conditional compilation allows the selective inclusion of sections of source text during compilation. Conditional compilation is controlled by the "B", "D" and "E" pseudo-comment directives and the boolean values associated with compile-time identifiers known as **compile flags.**

Compile flags are declared by using the Declare option before the program or unit heading. The pseudo-comment directive "D" is followed by a unique flag name which must conform to the same syntax as a UCSD Pascal identifier (see section 3.11.1). The initial value of the flag is set by a trailing "+" (indicating TRUE) or "-" (indicating FALSE). In the absence of a trailing "+" or "-", the flag value defaults to TRUE. Compile flag values may be redefined within the program. Attempts to redefine flags not declared before the beginning of the program or unit generate a syntax error.

Examples of compile flag declaration and value assignment:

```
{$D debug}    — Declare flag "debug" with value TRUE
{$D Z80+}     — Declare flag "Z80" with value TRUE
{$D list-} — Declare flag "list" with value FALSE
{$D debug↑}   — Set flag "debug" to its prior value
```

Source code is selectively included in a compilation by using the Begin and End options. These are analogous to BEGIN and END in Pascal. When the compiler scans a "B" pseudo-comment directive which contains a valid compile flag, the value of the flag expression determines whether the source text between the "B" directive and its corresponding "E" directive is to be compiled. If the flag expression evaluates to FALSE, the compiler skips over source text until it encounters an "E" directive containing the flag identifier. If the flag expression evaluates to TRUE the source text is included in the compilation.

If the compile flag in the "B" directive is followed by a "-" switch, the flag expression is equal to the logical negation of the flag identifier value.

NOTE: The "^" switch is ignored if it appears in the flag expression of the "B" directive. All switches are ignored in the "E" directive, but are useful for documentation purposes.

WARNING: Unit interface text is stored in a library without regard for the values of embedded flag expressions. Thus, conditional compilation directives can be found in imported interface text. All such flags should be defined before the beginning of a host that uses the unit. For added security, it is recommended that the value of the flag be redefined in the interface text so as to avoid any inconsistency between the unit's actual interface and the interface perceived by the host.

Example of conditional compilation:

```
{$D debug-}       {Declare flag "debug" with value FALSE}
program demo;
begin
  {$B debug}     {The following statement is not compiled}
  writeln ('there is a bug');
  {$E debug}

  {$D debug+}    {Set debug to TRUE}

  {$B debug}     {The following statement is compiled}
  writeln ('now there really is a bug');
  {$E debug}

  {$D debug↑}    {Restore debug to FALSE}
end {demo}.
```

5.0.6 I/O Checks

The compiler normally emits I/O checks after every file I/O operation. These checks cause an execution error if the I/O result (see section 3.11.9) reveals that an I/O error occurred during the operation.

I/O checks are emitted when the I/O Check option is enabled. The I/O Check option is controlled by the pseudo-comment directive "I" used as a switch option.

The default setting of the I/O Check option is on. "I-" disables the option and suppresses the generation of I/O check code. I/O checking may be restricted to portions of a program by selectively enabling ("I+") and disabling ("I-") the I/O Check option.

NOTE: Programs compiled with the I/O Check option disabled require that the I/O result be checked explicitly. Failure to provide these checks leaves a program susceptible to unexpected actions of both a human and mechanical nature.

Example of I/O check directives:

```
{$I+}
{$I-}
{$I↑}
```

See section 3.11.9 for more information. An example using I/O check directives appears in section 6.9.

5.0.7 Range Checks

The compiler normally emits range checks before every indexed array reference or subrange assignment. These checks cause an execution error if an array is indexed outside of its declared bounds, or if a subrange variable is assigned a value outside of its declared range.

Range checks are emitted when the Range Check option is enabled. The Range Check option is controlled by the pseudo-comment directive "R" used as a switch option.

The default setting of the Range Check option is on. "R-" disables the option, and suppresses the generation of range check code. Range checking may be restricted to portions of a program by selectively enabling ("R+") and disabling ("R-") the Range Check option.

NOTE: Programs compiled with the Range Check option disabled are smaller and faster than their cautious counterparts. However, they must be correct at the outset, for undetected range errors can propagate various and sundry species of nasty and elusive bugs. Proofs of program correctness are left to the user.

NOTE: p-System releases prior to Version IV.13 did not permit disabling the Range Check option in the case of bad indexes into string variables. The following code will generate a range error in a pre-IV.13 environment but not in a IV.13 or later environment.

```
var
  s: string;
begin
{$R-}
  s := 'hi';
  write(s[3]); {dynamic length is 2; program can bomb here}
{$R↑}
```

NOTE: The Range Check option only affects the generation of the execution error "Value range error" in the cases mentioned above. The I/O Check option affects the generation of execution errors due to I/O faults. In Version IV.1 and later releases of the p-System these error conditions may be trapped using the system ERRORHANDLER unit.

NOTE: In Version III implementations, if the second argument to the MOD operator is negative a nonsuppressable value range execution error occurs.

NOTE: Apple Pascal provides range checking that assures that the maximum length of a formal VAR string parameter equals or exceeds the maximum length of the corresponding actual parameter (see section 3.4.1 for a discussion of this issue). This range check is toggled using the "V" switch option, which has a default value of "V-" (no range check).

Example of range check directives:

```
{$R+}
{$R-}
{$R↑}
```

An example using range check directives appears in section 6.4.2.

5.0.8 GOTO Restriction

Pre-Version IV releases of the p-System include the "G" switch option to enable or prevent the programmer from using the GOTO statement. The default value is "G-"; GOTOs are not allowed. The intent is to discourage student programmers from indiscriminately using the GOTO instead of the preferred structured constructs.

Version IV ignores the "G" option. GOTOs are always permitted.

5.0.9 Copyright Notices

Copyright notices (or other textual information) may be embedded in a program's code file with the Copyright option. The notice is placed in block 0 of the code file (see your System Architecture Guide for details).

The Copyright option is controlled by the pseudo-comment directive "C" used as a string option. The string may contain up to 77 characters.

Copyright notices may be embedded in an already-compiled code file using the Library utility.

NOTE: Copyright directives must appear before the program or unit heading.

Example of copyright directive:

```
{$C copyright (c) 1982 by SurfDreck MondoSystems, Inc.}
```

5.0.10 Console Display Suppression

The compiler normally displays a running account of its progress on the console screen. Enabling the Quiet option suppresses the console display, resulting in faster compilations (due to the time saved by not writing to the console).

The console display is suppressed when the Quiet option is enabled. The Quiet option is controlled by the pseudo-comment directive "Q" used as a switch option.

The default setting of the Quiet option is off. "Q+" enables the option and suppresses the console display. The display may be restricted to portions of a compilation by selectively enabling ("Q+") and disabling ("Q-") the Quiet option.

NOTE: A field may be set in the SYSTEM.MISCINFO file to indicate that a system has a slow terminal. In that case the default for the "Q" compile option would be "Q-" rather than "Q+". See your Installation Guide for instructions on configuring SYSTEM.MISCINFO.

Example of quiet compile directives:

```
{$Q+}
{$Q-}
```

5.0.11 Segment Residency

In Version IV of the p-System UNITs are treated as segments which may be dynamically loaded and unloaded from memory by the operating system.

In pre-Version IV releases, however, segment procedure code is normally resident only during its execution but unit code is resident throughout the program's execution. In Apple Pascal, the default behavior for segment residency may be altered by using the Noload and Resident compile options.

The Noload option allows used units to be swapped as if they were segment procedures. It is controlled by the pseudo-comment directive "N" used as a switch option. The directive appears at the beginning of a host program. "N+" permits unit code to be swapped out of memory when inactive. "N-", the default value, forces unit code to be resident as long as the host program is active.

The Resident option allows segments and/or swappable units to be memory-resident throughout the execution of a given procedure. Segment residency is controlled by the pseudo-comment directive "R" used as a string option. The directive appears immediately after the first BEGIN (and before the first statement) of the desired procedure and contains a list of segments and swappable units to be made memory-resident. Segment and unit identifiers are separated by commas, and spaces are ignored.

NOTE: A Resident option applied to a segment or unit that is already memory-resident has no effect.

NOTE: Misplaced Noload and Resident options are ignored.

NOTE: The MEMLOCK and MEMSWAP intrinsics may be used to control segment residency in Version IV. See section 3.1.1 for details.

Example of segment residency directives:

```
program favoritethings;
uses raindrops, roses, sashes;
 {$N+}                     {all segments are swappable}

  segment procedure snowflakes;
  begin
  end {snowflakes};

  procedure music; {snowflakes and raindrops remain}
  begin            { resident throughout call to music}
    {$R snowflakes, raindrops}
    snowflakes;
  end {music};

begin
  music;
end {favoritethings}.
```

5.0.12 System Programs

The p-System itself is written in UCSD Pascal. This not only includes the various utilities available under the p-System, but also the operating system, SYSTEM.PASCAL. Access to certain facilities, such as system variables, which are normally accessible only to systems programs is occasionally required by application programs. The User Program compile option allows the programmer to inform the compiler whether or not these additional facilities need be provided.

It should be pointed out that programs compiled as system-level programs behave in a markedly different manner from user-level programs. The programmer should become thoroughly familiar with all the ramifications of specifying the System level before attempting to utilize it.

Programs are normally compiled to execute at the level defined for user programs. The User Program option may change the level to that of the operating system.

A program is compiled at the system level when the User Program option is disabled. The User Program option is controlled by the pseudo-comment directive "U" used as a switch option. The default setting of the User Program option is on. "U-" disables the option and also sets the following options: "G+", "R-" and "I-".

NOTE: The User Program option directive must appear before the program heading. Unlike other options, User Program may not be selectively enabled and disabled during compilation.

There are drastic Version-dependent differences in the treatment of programs compiled at the System level. We will discuss first how these are treated under pre-Version IV releases of the p-System, then how these are treated under the current Version IV release.

5.0.12.1 System Programs Before Version IV

In UCSD Pascal, blocks of code may be nested within other blocks of code. Programs or units contain segments, segments may contain additional procedures, some of which may themselves be segments, and so on. For example, GETCMD is a segment nested within the UCSD Pascal operating system. The term "lexical level" is used to indicate the depth within which one block of code in the UCSD Pascal system is nested within another.

The operating system itself is at the "outer" lexical level, since it is the block of code containing all the others. This outer lexical level is assigned a level number of -1. As the depth of nesting increases, the lexical level increases by one. GETCMD, which is a segment in the operating system, is at lexical level 0. A procedure within GETCMD would be at lexical level 1; a procedure within that procedure would be at lexical level 2, and so on.

A user program is considered to be nested within the operating system at lexical level 0. The operating system, when compiled, has fixed "slots" for the various segments within it which are at lexical level 0. GETCMD occupies one such slot. A number of such slots are left empty; these are reserved for segments and units of the current user program.

The compiler normally compiles programs to execute at the lexical level defined for user programs, level 0. The program itself occupies slot 1 in the operating system. Any segments it uses occupy slots 7 through 15 (10 through 15 on early releases of Version II).

When the "U" switch option is used to inform the compiler that a program is to be compiled at the System lexical level, the compiler treats the program as a "pseudo-operating system". However, the program itself *must not contain any code.*This is because the program is not the operating system, merely a user program compiled at the System level. If there were code in the program, it would be at lexical level -1. Instead, the program should have a "dummy" outer lexical level; the body of the program should consist of an empty BEGIN/END pair:

```
begin
end.
```

The actual program should be included as a segment procedure following the declarations. This segment procedure will be compiled at lexical level 0, since it is nested directly within the pseudo operating system, and will occupy slot 1 since it is the first segment encountered by the compiler. (Were there code in the main program itself, it would occupy slot 0.)

The segment procedure for the actual program may contain as many functions and procedures as are required (up to the limit of the p-System implementation, of course). If additional segment procedures are required, they should be contained within the "main" one, subject to the sequence requirement described below.

User program segments beyond the main segment normally occupy slots beginning at slot 7 in the operating system. The compiler automatically generates slot numbers beginning at 7 for these additional segments if the program is at the User lexical level, but will generate consecutive slot numbers beginning at 2 if the program is at the System lexical level. Therefore, to assure that the segments within the System level program occupy the same slots they would normally occupy within the real operating system, the system-level program must contain a number of dummy segment procedures to fill the intervening slots. Since the "main" segment is in slot 1, five dummy segments are required before beginning the real segments. They should take the following form:

```
segment procedure dumslot2;
begin
end.
```

This will force the necessary (non-dummy) segment procedures to occupy slots beginning at slot 7.

There are a number of additional considerations when compiling programs at the System lexical level. Under most implementations, the level 0 segment procedure (the one containing all the code of the program) must have two dummy parameters, both integers. Forward declared procedures may be unresolved. As mentioned previously, the $R-, $I- and $G+ compiler options are automatically set when $U- is set.

The most important consideration, however, is that programs compiled at the system level may access system variables and procedures normally outside the scope of user programs. The system global variables include those that define terminal characteristics, default and prefixed volumes, system date and many other type, variable and forward declarations.

The variables declared in the System level program at the outermost level are caused by the compiler to overlay the area in memory where the system globals are stored. Of course, the declarations must correspond exactly to the declarations of the actual system globals. The usual method of assuring that this is the case is to Include (section 5.0.2) a file containing the global declarations in the exact form in which they appear within the operating system. (Many systems come with a file called GLOBALS.TEXT; the globals for most p-System implementations are available from the library of the UCSD Pascal Users' Society.)

WARNING: Attempts to execute system-level programs on systems utilizing different globals than were used to compile the program result in mysterious and fatal system crashes.

NOTE: Code files generated for system level programs contain an extra block due to the emission of a dummy segment for the pseudo-operating system level. This block is automatically removed when the Library program (described in the System User's Manual) is used to create a copy of the code file.

Example of pre-Version IV user-level directive use:

```
{$U-}
program fakeOS;
  {$I GLOBALS.TEXT}
  {   These are the declarations corresponding to the
      operating system global variables }

  segment procedure userprogram(dummy1, dummy2: integer);
  var progglobal: integer;  { or whatever }

    segment procedure dumslot2; begin end;
    segment procedure dumslot3; begin end;
    segment procedure dumslot4; begin end;
    segment procedure dumslot5; begin end;
    segment procedure dumslot6; begin end;

    segment procedure progseg;
    begin
```

```
      {the code goes here}
    end {progseg};

    procedure progproc;
    begin
      {the code goes here}
    end {progproc};

  begin
    {the code goes here}
  end {userprogram};

begin {should contain NO code}
end {fakeOS}.
```

5.0.12.2 System Programs Under Version IV

Under Version IV the "U-" compile option should never be used, except by operating system-level programs.

A program compiled as a system-level program will appear identical to an ordinary program. There are no lexical level or operating system slot considerations. Variables do not overlay the system globals.

The I/O check and range check compiler options must be set explicitly within the user program (there is no GOTO compiler option in Version IV). Certain names reserved by the system, such as the the names of the operating system segments, may be used in a program compiled with the "U-" option.

Programs that must make use of the the system globals may do so cleanly, by using a UNIT provided with Version IV for that purpose. The KERNEL unit of the operating system contains these globals; its interface section is available in the library KERNEL.CODE. Programs may use this unit in the ordinary manner (see section 5.0.3).

System units and data structures are discussed in further detail in chapter 8.

5.0.13 Native Code

Version IV.1 of the p-System introduced the capability to translate selected procedures of a Pascal program from p-code to the native machine code of the host processor. This provides an increase in speed at the cost of larger code size.

The translation is accomplished after the program is compiled by submitting the code file to a utility called the Native Code Generator (NCG). The compiler flags the procedures which are to be translated by the NCG into native code; the NCG produces a faster (but larger) code file which will run only on the single processor for which the NCG was designed. There is a version of the NCG for many major processors.

The native code option is controlled by the pseudo-comment directive "N" used as a switch option. The default setting of the native code option is off (N-). Native code generation may be enabled by placing the native code directive N+ before the first BEGIN of a procedure. It may be disabled after the last end of the procedure. Native code generation may only be performed on entire procedures or collections of procedures.

NOTE: The p-code file produced by the compiler from program source containing the "N+" option is executable as-is on any processor. It is slightly larger than the equivalent file without the "N+" option, since the compiler must generate additional p-code for use by the NCG.

Example of native code generation:

```
program lotsawork;
procedure loopy;
var i: integer;
{$N+}
begin
  for i:= 1 to 32000 do
    begin
      {many detailed things}
    end;
end;
{$N↑}
end.
```

Native code is discussed in further detail in section 6.13.

5.0.14 Real Size

Pre-Version IV releases of the p-System uses two words to store real values in code files. The representation is not always portable between p-System implementations. A goal of the Version IV release was to move toward a standard, four-word representation of real values.

However, the Version IV release still supports two-word reals. The compiler generates code to handle either two- or four-word reals based on the real size compiler options. This option must be specified at the beginning of a program and cannot be changed within the program.

The R2 button directive specifies that the compiler should generate code for two-word reals; the R4 button directive is used for four-word reals.

The default value depends on the interpreter in use and can be determined by examining the title line appearing on a compiler listing or the compiler heading on the console. The digit of the compiler version number following the dash (-) should be either a 2 or a 4, and specifies whether two- or four-word reals is the default. For example, the following title indicates a default to two-word reals:

```
Pascal Compiler IV.13 c6t-2
```

NOTE: Although the compiler generates a code file using whatever real size is specified, the resulting code file will not execute unless the corresponding two- or four-word REALOPS unit is installed in the operating system, and the interpreter is configured with the corresponding real size. See your Installation Guide for details. Attempts to execute a program which uses real values on a mismatched interpreter or operating system results in execution error 17 ("Incompatible real number size").

5.0.15 Symbolic Debugging

Symbolic debugging, introduced in Version IV, allows the programmer to stop execution of a program at selected points and determine status information using variable identifiers before resuming execution. For example, a programmer may wish to determine the value of a variable using its declared name at various points during the programs execution.

The compiler does not normally include identifier names in a code file. Since it must do so to make symbolic debugging possible, the "D" switch option may be included to bracket those portions of a program for which the compiler must include identifiers.

The D+ option is used to cause the compiler to begin including identifier names; the D- option is used to tell the compiler to stop including them.

Use of the D+ option may significantly increase the size of the code file.

Further detail regarding use of the Debugger utility may be found in the System Users' Manual.

5.0.16 Byte Sex Flipping

The format of a word within memory may differ from one processor to another. On some machines, the low-order byte of a word may occupy an even address, while the high order-byte occupies the following odd address.

On other machines, the high-order byte occupies the even address, while the low-order byte occupies the following odd address.

This distinction is referred to as "byte sex". In pre-Version IV releases of the p-System, the compiler generates code corresponding to the byte sex of the host machine. This code cannot execute on a machine of the opposite byte sex.

The "F" switch option is provided in these versions to enable compilation of code which would run on a machine of whichever byte sex was selected. F+ enables code generation for a machine where the high-order byte falls on the even address. F- enables code generation for a machine where the low-order byte falls on the even address. The default is the byte sex of the host machine.

The byte sex compiler option must appear at the beginning of a program and cannot be changed within a program.

The byte sex option is unnecessary in Version IV because although the compiler generates code of the host's byte sex, the operating system allows code of either byte sex to execute. If the code file is not of the native byte sex, the system automatically converts the code as it is executed.

5.0.17 Executable Units

P-System code files may contain programs, units or both. Programs are designed to stand alone while units are designed to be used by programs or other units. However, the internal structure of programs and units is very similar. In fact, the only internal difference between a unit code file and a program code file is a flag set to indicate which is which. See section 8.4.2 for details of code file format.

Both types of code files have a primary segment. In a program code file, the primary segment contains the outer level code of the program. In a unit code file the primary segment contains the unit's initialization and termination code.

It is sometimes convenient to permit a code file to serve both as a unit, useable by other programs, and as a program which can be executed by itself. This is accomplished by using the unit syntax, but compiling with the H+ switch option. A code file produced in this manner may be used as an ordinary unit. It may also be used as a program by executing it in the usual manner.

Use of host units is particularly convenient when a used unit must utilize procedures of the host program. Normally, this is not permitted since the identifiers of the host program are not known to the unit. If the *host* is compiled as a unit with the host unit option, however, it may be used by the same unit it is using! That is, the two units will use each other via a circular USES, with one unit acting as the main program. This is perfectly legal and often quite convenient.

When a unit compiled with the host unit option is used as a program, execution begins with the initialization code. The initialization code may freely call upon the functions and procedures within the unit, including those that appear in the INTERFACE section and those that do not. Note that a unit compiled with the host unit option may not contain termination code.

The host unit compiler option must appear at the beginning of the unit and cannot be changed. The default value is H-, which causes the unit to be compiled as an ordinary unit.

```
{$H+}
unit ProgUnit;
interface
  procedure p1(s: string);
implementation
uses {$U UNITUNIT.CODE} UnitUnit;
  var
    locstring: string;
  procedure p1;
  begin
    writeln(s);
  end;

begin {initialization section/program body}
  writeln(
    'Hello there from initialization section of ProgUnit');
  locstring := 'Off we go to p2, in UnitUnit';
  p2(locstring);
  {NOTE: no termination section allowed with $H+}
end.


unit UnitUnit;
interface
  procedure p2(s: string);
implementation
uses {$U PROGUNIT.CODE} ProgUnit;
  var
    locstring: string;
  procedure p2;
  var
    s2: string;
  begin
    writeln(s);
    s2 :=
     'And now back to p1 in ProgUnit from p2 in UnitUnit!';
    p1(s2);
  end;
end.
```

In this example `ProgUnit` is the unit serving as a "program"; it is compiled with the H+ switch option and contains initialization code that is executed when `ProgUnit` is run. `ProgUnit` writes an identifying message, then passes a string to procedure `p2` in `UnitUnit`.

`UnitUnit` is an ordinary unit, compiled without the H+ option. It can only be used in conjunction with a host. Procedure `p2` in `UnitUnit` writes the string passed to it by `ProgUnit`. Then, `p2` sets up its own string and calls on procedure `p1` in `ProgUnit` to write that string out. Thus, `ProgUnit` uses `UnitUnit` and `UnitUnit` uses `ProgUnit`.

The tricky part of this illustration is compiling the two units. Each unit must contain a USES indicating the usage of the other unit. The question is which unit to compile first; if `ProgUnit` is compiled first the compiler will generate an error message since it will not be able to find UNITUNIT.CODE. If `UnitUnit` is compiled first the compiler will not be able to locate PROGUNIT.CODE.

One solution to this dilemma is to prepare a version of one of the units – `ProgUnit`, for example – with the interface section intact but with all references to `UnitUnit` removed, as illustrated:

```
unit ProgUnit;
interface
  procedure p1(s: string);
implementation
  procedure p1;
  begin
  end;
end.
```

This "phony" `ProgUnit` is compiled to a code file called PROGUNIT.CODE. Then `UnitUnit` is compiled in the usual manner. The compiler finds and includes the interface section of the "phony" `ProgUnit`, which is identical to the interface of the true `ProgUnit`. Then the true `ProgUnit` is compiled to the code file PROGUNIT.CODE. The compile procedes without error since UNITUNIT.CODE already exists. The new PROGUNIT.CODE replaces the earlier "phony" version.

`ProgUnit` may now be executed. The output is:

```
Hello there from initialization section of ProgUnit
Off we go to p2, in UnitUnit
And now back to p1 in ProgUnit from p2 in UnitUnit!
```

5.0.18 "Tiny" Compiler

The Version II Pascal compiler contains a number of devices to reduce the amount of space required by the compiler during its execution. The less space consumed by the compiler, the larger the program that can be compiled.

One of these devices is the "T" compiler switch directive. The effect of T+ is to effectively remove a number of predefined procedures from UCSD Pascal, thereby reducing the compiler's memory requirements. Of course, the T+ directive may only be used in programs that do not use any of these predefined procedures.

The procedures removed by T+ are READLN, PRED, SUCC, SQR, UNITREAD, INSERT, DELETE, COPY, POS, SEEK, GET, PUT, PAGE, STR and GOTOXY. The choice of procedures to remove is governed by the needs of the various Version II systems programs that cannot compile without T+. In particular, the Version II Pascal compiler itself cannot compile on many machines without the T+ directive.

5.1 Option Summary

B	Starts a conditional compilation section based on the following flag expression.
C	The following string is embedded in the code file as a copyright notice.
D	"D+" enables symbolic debugging. "D–" disables symbolic debugging. Default value: "D–"
D	Declares/defines a conditional compilation flag and sets its value. Default value: TRUE
E	Terminates a conditional compilation section based on the following flag.
F	"F+" generates code for even byte sex. "F–" generates code for odd byte sex. Default value: processor–dependent
G*	"G+" enables use of GOTO. "G–" disables GOTO. Default value: "G–"
H	"H+" enables use of a unit as a standalone program.

	"H-" permits the unit to be used only as a normal unit. Default value: "H-"
I	"I+" generates I/O checks after file I/O operations. "I-" suppresses checks. Default value: "I+"
I	The following string contains the name of a text file to be "included" into the source.
L	"L+" enables listing. "L-" suppresses listing. Default value: "L-"
L	The following string contains the name of the compiled listing file.
N	"N+" flags procedures for native code generation. "N-" disables native code generation. Default value: "N-"
N*	"N+" causes UNIT code to be resident only when active. "N-" causes UNIT code to be resident as long as host program executes.
P	"P" emits a page break in listing.
Q	"Q+" enables the console display. "Q-" suppresses the display. Default value: "Q+"
R	"R+" generates range checks on array indices and subrange variables. "R-" suppresses checks. Default value: "R+"
R*	The following segment/unit list is to be made resident throughout the current procedure.
R2	Generate code for two-word reals. Default value for R2 or R4: interpreter-dependent
R4	Generate code for four-word reals. Default value for R2 or R4: interpreter-dependent
S*	"S+" specifies swapping compiler. "S++" specifies more swapping.

	"S-" specifies no swapping. Default value: "S-"
T*	"T+" specifies "tiny" compiler. "T-" specifies normal compiler. Default value: "T-"
T	List the following title on every succeeding page of the compiler listing.
U	"U+" specifies a user-level program. "U-" specifies a system level program and "G+,I-,R-". Default value: "U+"
V*	"V+" specifies to perform a range check to assure that the maximum length of a VAR string parameter meets or exceeds the maximum length of the corresponding actual parameter. "V-" specifies not to perform such checking. Default value: "V-"

NOTE: An asterisk (*) indicates options that are ignored or not available under Version IV.

Chapter 6

PROGRAMMING PRACTICES

Contents

6.0 Packed Variables and Storage Allocation 170
6.0.1 Packed Arrays 171
6.0.2 Packed Records 173
6.0.3 Packing and Storage Allocation Rules 174
6.1 Accessing Bytes, Bits and Bit Fields 178
6.1.1 Words 178
6.1.2 Bit Fields 179
6.1.3 Bits 180
6.2 Unsigned Integer Manipulation 182
6.3 Full-word Logical Operations 186
6.4 Variable-sized Array Allocation 186
6.4.1 Version II Heap Strategy 187
6.4.2 Version IV Heap Strategy 189
6.5 Segment and External Procedures in Unit Interfaces 191
6.6 Structured Parameters Using Pointers 192
6.6.1 Technique 193
6.6.2 Heap Management 195
6.7 Passing "Untyped" Parameters 196
6.8 Variant Record Buffer Overlay 201
6.9 Data Prompts 203
6.9.1 Character Prompts 204

6.9.2 Integer Prompts 204
6.9.3 File Prompt 205
6.9.4 Real Prompts 206
6.10 Device Drivers 210
6.10.1 Driver Interface 211
6.10.2 Device Access 212
6.11 Locating Execution Errors 215
6.11.1 Using Compiled Listings 215
6.11.2 Without Using Compiled Listings 216
6.11.3 Further Investigations 217
6.12 Programming with Units 217
6.12.1 Unit Development 217
6.12.2 Using Pre-existing Units 219
6.13 Using Native Code 221
6.13.1 Automated Native Code Generation 221
6.13.2 User-Supplied Native Code 223
6.14 Passing Parameters Between Programs 223
6.15 Coding Style and Optimizations 226
6.15.1 Expressions and Array Indices 226
6.15.2 Multiword Constants 227
6.15.3 Packed Field References 228
6.15.4 Reals and Long Integers 230
6.15.5 Short Forms 231
6.15.6 WITH Statements 234
6.15.7 String Manipulation 236
6.15.8 CASE Statements 238
6.15.9 GOTO Statements 239
6.15.10 Procedure Calls 240
6.15.11 Parameters to Procedures 241

This chapter describes common UCSD Pascal programming practices. Note that most of these practices are implementation dependent - they should not be used in programs intended for use outside of the UCSD Pascal system.

Section 6.0 describes packed variable allocation (knowledge of which aids the design of compact data structures). Section 6.1 explains how to access arbitrary words, bit fields and bits in memory. Section 6.2 explains how to perform unsigned integer arithmetic and comparisons. Section 6.3 explains how to perform logical operations on word quantities (e.g., integers). Section 6.4 shows how the UCSD Pascal heap implementation can be exploited to create dynamic arrays. Section 6.5 discusses a current limitation regarding segment procedures, and suggests a means of working around that limitation.

Section 6.6 demonstrates a technique for allowing a function to take on and to return a value of a structured type. Section 6.7 details a means of passing a parameter to a subroutine without requiring that the subroutine know the type of the parameter being passed. Section 6.8 explains how to perform I/O between byte streams and buffers, and overlay one or more record structures onto the buffer. Section 6.9 describes the implementation of data prompts in a manner suitable for an interactive environment. Section 6.10 explains how to write asynchronous device drivers in UCSD Pascal. Section 6.11 describes methods for locating execution errors in Pascal programs. Section 6.12 describes how, why and when to use separate compilation units. Section 6.13 describes the benefits and pitfalls of using native code generation. Section 6.14 describes a means of transmitting data between CHAIN'ed programs. Finally, section 6.15 discusses programming styles for UCSD Pascal that lead to efficient, optimized code.

6.0 Packed Variables and Storage Allocation

This section describes the implementation of packed variables in UCSD Pascal and the algorithms used by the compiler to allocate storage for packed and unpacked data items.

There are two reasons a high-level language programmer may wish to consider these low level issues. The first reason is to construct data structures consuming minimal amounts of space. The second reason is to map data originating from a "foreign" source to an appropriate Pascal data structure. For example, a programmer may wish to read information from a CP/M disk into a UCSD Pascal record, or accept data following a communications protocol from a remote port.

Normal programming procedure is to first describe the data structure in Pascal in a form convenient for use within a program. The data structure will then determine the binary representation. However, in situations where the file or record structure are unrecognized by the p-System the situation is reversed. The programmer is forced to construct a Pascal record whose binary representation corresponds to the already existing external data.

Record and array data are stored in a packed representation when their type declaration is preceded by the reserved word PACKED. Packing is not performed on files and sets. (Note that since sets are represented as bit strings in UCSD Pascal, with one bit for each value in the set's base type, specifying a set as packed is unnecessary and thus ignored. A file is stored as a "back-to-back" sequence of records, each corresponding to the internal representation of the file's base type. Thus, packing files is also redundant.)

Individual data items (items that are not elements of arrays or records) are not affected by packing; they occupy the same amount of storage whether packed or not.

Variables that have been declared as PACKED may be used as ordinary variables. The compiler automatically generates code to perform unpacking (and repacking) of a variable as it is accessed. Thus, the Standard Pascal procedures PACK and UNPACK are unnecessary and unavailable in UCSD Pascal. The exception is that packed fields may not be passed as VAR parameters. The programmer must assign the field to an unpacked variable, pass that as the VAR parameter, then assign the returned value back to the packed field.

A good candidate for packed format is data that occupies large amounts of space, but which is accessed relatively infrequently. A decision to use packed data should be influenced by two distinct tradeoffs: speed versus space, and code space versus data space. The speed-versus-space tradeoff is the space saved by compressing the data representation versus the slower access time (caused by packing and unpacking data during every variable reference). The code-versus-data tradeoff is the increase in program size (caused by extra code for packing and unpacking data at every variable reference) versus the space saved by compressing the data representation. Note that the latter tradeoff is static; the extra code space is the sum of the code generated for each of the variable references contained in a program, and will not change from one execution of the program to the next. The former tradeoff is dynamic; it depends on how many variable references are executed by a program, and may change from one execution of the program to another.

Sections 6.0.1 and 6.0.2 present examples of packed arrays and records respectively. Section 6.0.3 presents the packing rules and restrictions for UCSD Pascal.

NOTE: The SIZEOF intrinsic is useful for determining the size of a packed type. See section 4.37 for details.

6.0.1 Packed Arrays

UCSD Pascal performs packing of arrays if the array type definition is preceded by the reserved word PACKED. Consider the following type definitions:

```
type
    large = array[0..9] of char;
    small = packed array[0..9] of char;
```

Character variables are normally allocated a full word, but can fit in a single byte. A variable of type `large` is allocated ten words of data space and each character element is allocated a full word for storage. A variable of type `small` is allocated five words of data space. Two character elements are packed into each word and each element is allocated a single byte for storage.

Examples of packed arrays:

```
type
    one = packed array [0..7] of 0..3;
    two = packed array [0..2] of 0..31;
    zip = packed array [0..2] of set of 0..8;
```

Variables of type `one` fit in one word. Each element occupies two bits, since all values in the range 0..3 can be expressed in two bits. Hence the eight elements fit into a single word. Variables of type `two` are also one word long. Each element is five bits long, and three of them fit in a word (with the high order bit unused). Variables of type `zip` require three words, one for each element of the array. The UCSD Pascal packing algorithm does not pack consecutive array elements into a single word unless the elements are each no more than eight bits long and fit entirely within a word (see section 6.0.3). Since the base type of `zip` is nine bits long, each element is allocated a full word (9 bits for the set, 7 bits unused).

The following type definitions are not equivalent in UCSD Pascal:

```
type
    a = packed array[0..5] of array[0..7] of char;
    b = array[0..5] of packed array[0..7] of char;
```

Type definitions containing nested arrays are packed only if the last occurrence of the reserved word ARRAY is preceded by PACKED. In the example above, packing is performed on variables of type `b`, but not on variables of type `a`. To ensure packing of types containing mixes of arrays and records, precede all occurrences of ARRAY and RECORD with PACKED.

NOTE: String constants are type-compatible only with packed character arrays; they are incompatible with unpacked character arrays.

6.0.2 Packed Records

UCSD Pascal performs packing of records if the record type definition is preceded by the reserved word PACKED. Consider the following type definitions:

```
type large = record
               a,b,c,d: char;
             end;
     small = packed record
               a,b,c,d: char;
             end;
```

Character variables are normally allocated a full word, but can fit in a single byte. A variable of type `large` is allocated four words of data space and each character field is allocated a full word for storage. A variable of type `small` is allocated two words of data space. Each field is allocated a single byte for storage, so two character fields are packed into each word.

Examples of packed records:

```
type
     one = packed record
             f1,f2,f3,f4: 0..3;
             byte: 0..255;
           end;
     two = packed record
             f1,f2,f3: 0..31;
           end;
     zip = packed record
             f1,f2,f3: set of 0..8;
           end;
```

Variables of type `one` fit in one word. Each `f` field is two bits long and the four fields fit into a single byte. The `byte` field occupies the other byte in the word. Variables of type `two` are one word long. Each field is five bits long, and three of them fit in a word (with one bit unused). Variables of type `zip` require three words. Packing is not performed because the fields are larger than 8 bits; as with arrays, the UCSD Pascal packing algorithm does not pack separate record fields into a single word unless the fields are each no more than eight bits long and fit entirely within the word (see section 6.0.3). So each field in `zip` is allocated a full word (9 bits for the set, 7 bits unused).

The following type definitions are *not* equivalent in UCSD Pascal:

```
type
     a = packed record
           i: integer;
           r: packed record
                r1,r2: char;
              end;
```

```
        end;
    b = packed record
           i: integer;
           r: record
                r1,r2: char;
              end;
        end;
```

Type definitions containing nested records are packed only if the innermost occurrence of the reserved word RECORD is preceded by PACKED. In the example above, packing is performed on variables of type `a`, and the fields `r1` and `r2` will occupy the same word. But packing will not be performed on variables of type `b`. The fields `r1` and `r2` will occupy separate words. To ensure packing of types containing mixes of records and arrays, precede all occurrences of ARRAY and RECORD with PACKED.

NOTE: When a record contains a variant part, it is allocated enough space to contain the largest variant (unless dynamically allocated with NEW(<pointer>,<variant list>)).

6.0.3 Packing and Storage Allocation Rules

This section describes the rules for the allocation of storage and packing of variables in UCSD Pascal. These include constraints imposed on variable packing and optimizations performed within those constraints.

The following table displays packed and unpacked sizes for some common types:

Type	Unpacked	Packed
integer	word	word
boolean	word	1 bit
char	word	8 bits
real	2 or 4 words	2 or 4 words
subrange a..b	word	log base 2 (b − a) bits
set 0..n : n<16	word	n bits
long integer	implementation dependent	
enumerated, n elements	word	log base 2 (n + 1) bits

NOTE: Subranges are packable only if their range values are nonnegative. If either bound is negative, they are not packed.

With the exception of sets that occupy less than a word, structured fields (i.e., nested records or arrays within records or multiword sets within records) must begin on word boundaries, and are thus non-packable.

The primary constraint limiting packing within records and arrays is that fields in packed variables cannot be packed across word boundaries. Therefore, records benefit from packing only if they contain a number of scalar, subrange or set fields, each of which can be stored in 8 bits or less, and these fields are stored consecutively. Consecutively declared fields needing more than 8 bits apiece cannot be packed because of the aforementioned restriction that variables within records cannot be packed across word boundaries. Fields needing more than 8 bits are allocated one or more words for storage, and are accessed as unpacked fields. Fields are generally allocated storage space in the order in which they are declared in a record. Thus, rearranging the fields in a record sometimes results in a smaller record, as fields can be packed only if they are declared adjacently to other packable fields (see the first example below).

NOTE: Isolated records and arrays occupy an integral number of words (an even number of bytes). Records whose components do not add up to an integral number of words are padded to the next word boundary.

NOTE: When a packable field is forced to occupy a full word because of adjacent word-aligned fields, it is accessed as an unpacked field.

NOTE: Packed records may contain both packed and unpacked fields – packing is determined by the sizes and declaration order of the record's fields.

Example of rearranging record fields:

```
type foon = packed record
              b1: boolean;     {1 word }
              i1: integer;     {1 word }
              b2: boolean;     {1 word }
              r1: real;        {2 words}
              b3: boolean;     {1 word }
            end;        {total = 6 words}

     newfoon = packed record
                 b1,b2,b3: boolean;  {1 word }
                 i1: integer;        {1 word }
                 r1: real;           {2 words}
               end;           {total = 4 words}
```

In `foon`, `i1` and `r1` are constrained to word boundaries because they occupy integral numbers of words (1 and 2 words respectively). The boolean fields are non-packable because of their adjacence to word-aligned fields. In `newfoon`, the adjacency of the boolean fields allows them to be packed into a single word (note that there is space in the word for up to 13 more (packed) boolean fields).

The word boundary restriction can often be defeated using variant records. Suppose, for example, it is necessary to read a "foreign" (i.e.,originating on a non-UCSD system) record consisting of three consecutive characters occupying one byte each, followed immediately by a single byte containing two hexadecimal digits. How may one declare a UCSD Pascal record to correspond to this data item?

The first impulse might be to declare the record as follows:

```
type
   record1= packed record
               three_chars: packed array[1..3] of char;
               hex        : packed record
                               hex1: 0..15;
                               hex2: 0..15;
                            end;
            end;
```

Unfortunately, this declaration is inadequate. `Three_chars` is the first field in `record1`, which begins on a word boundary. `Three_chars` is only 3 bytes long, however, so it does not end on a word boundary. The sub-record `hex` must begin on the next word boundary, leaving a one-byte "hole" between `three_chars` and `hex`. `Hex` itself occupies one byte, for a cumulative total of 5 bytes, but since records must occupy an integral number of words, the size of `record1` will be 6 bytes. `Record1` will therefore not correspond to the "foreign" data structure.

A better alternative might be as follows:

```
  record2= packed record
              case integer of
                1: (three_chars: packed array[1..3] of char);
                2: (junk: packed record
                            injunk1: integer;
                            injunk2: char;
                            hex1   : 0..15;
                            hex2   : 0..15;
                          end);
           end;
```

Here, `three_chars` appears as the first variant of `record2`, and occupies three bytes. In this instance, instead of following `three_chars` with the declarations for the two hex digits, a second variant called `junk` is constructed. The first two fields, `injunk1` and `injunk2`, correspond to the three bytes in the first variant. The sole purpose of these fields is to overlay `three_chars` but the overlay is broken into two separate fields so as not to violate the word boundary restriction. `Injunk1` occupies one word. `Injunk2` occupies only one byte, though, so both `hex1` and `hex2` (each requiring 4 bits) may be squeezed by the compiler into the byte immediately following `injunk2`. There are no "holes" in `record2`; its total size is 4 bytes. `Record2` corresponds precisely to the "foreign" data item.

NOTE: Fields in a packed record are normally allocated in the order of their appearance in the record declaration. The exception is a list of variables, separated by commas, which share the same type declaration. These variables are allocated in the reverse order of their appearance in the list. This scheme is called **reverse field allocation.**

Example of reverse field allocation in records:

```
type nibble = 0..15;
     widgetWord = packed record
                        switch:  0..1;
                        control: 0..7;
                        b1,b2,b3,b4: boolean;
                        device:  nibble;
                        unused:  nibble;
                     end;
```

`WidgetWord` is allocated in the following fashion: `switch` is in bit 0 (the least significant bit), `control` occupies bits 1-3, `b4` is in bit 4, `b3` in bit 5, `b2` in bit 6, `b1` in bit 7, `device` occupies bits 8-11, and `unused` occupies bits 12-15.

NOTE: An optimization to speed data access is applied by the compiler to packed records which contain up to 8 bits immediately before a word-aligned field. If this quantity is not already word-aligned, it is adjusted to occupy the entire byte preceding the word-aligned field which follows it, and is accessed as an 8-bit field.

Example of byte-alignment optimization:

```
type twimp = packed record
                b1: boolean; {starts on bit 0}
                ch: char;    {starts on bit 1, with no gap}
             end;

     newtwimp = packed record
                   b1: boolean;     {starts on bit 0}
                   ch: char;
                    {starts on bit 8, the byte preceding i}
                   i: integer;
                             {starts on next word boundary}
                end;
```

See section 6.1.2 (bit fields) for more information on variable packing.

6.1 Accessing Bytes, Bits and Bit Fields

This section presents methods enabling UCSD Pascal programs to access arbitrary words, bit fields and bits in memory. Very few programs require direct access to memory; it is generally restricted to system programs (see section 6.10 for an example).

NOTE: The methods outlined in this section are effective only for access to the stack and heap space. These methods are not effective for accessing the code pool or other memory areas.

NOTE: Memory access can also be accomplished with in-line machine code (see section 3.10). However, the methods presented here are less error-prone and do not require knowledge of the p-machine instruction set.

The following examples utilize the Standard Pascal feature known as a record variant. It is used here to provide a controlled form of type conversion, which in turn allows exploitation of the machine representations of various Pascal types for gaining direct access to the machine.

WARNING: The examples presented here are highly nonportable because of different system memory configurations and/or machine architectures (see your Architecture Guide for details).

6.1.1 Words

Example of arbitrary word access:

```
program c1;
type address = integer;
var  dest:     integer;

  function peek(location: address): byte;
  { return value at specified address }
  type trick = record case boolean of
                 true:  (addr: address);
                 false: (intptr: ↑integer);
               end;
  var  access: trick;
  begin
    access.addr := location;
    peek := access.intptr↑;
  end {peek};

begin
```

```
    write('Which word do you wish to examine? ');
    readln(dest);
    writeln('The word at memory address ',
            dest, ' has the value ', peek(dest));
  end {c1}.
```

In UCSD Pascal, pointers are maintained internally as one-word unsigned integers. It is possible, therefore, to treat an arbitrary memory location as a variable of any type by defining a pointer to that type and assigning to the pointer the address of the arbitrary memory location.

As in the example above, this may be done using a variant record. Variant records allow a variable to be accessed as any number of types. In the example above, `trick` is accessed first as an integer, then as a pointer. It is first assigned an integer value –the address of the memory location to be "peeked". Next it is used as a pointer in order to load the desired word.

Note that the variant record may be used to perform operations on data that are not normally allowed by the compiler. For example, it is possible to perform arithmetic on a pointer by assigning the result of an arithmetic expression to the integer variant.

One drawback of this method is the necessity of specifying large (i.e., > 8000 hex) addresses as negative integer values. For example, the word at address FFFF hex is returned by calling peek(-1). (A hexadecimal calculator such as the TI Programmer or HP 16C is useful in these situations.) A second drawback is the specification of address 8000 hex. It cannot be represented as an integer constant because its decimal value is -32768, which is treated by the compiler as a long integer constant. 8000 hex can be specified by the integer expression 32767 + 1 (remember: no integer overflow checks in UCSD Pascal).

6.1.2 Bit Fields

Bit fields are fields which occupy less than a byte. These are accessed in a similar fashion to bytes; by using packed records. The pointer variant is set to point to a one-word packed record, which is declared so that the desired bit fields are accessible. See section 6.0 for details on record packing.

Example of arbitrary bit field access:

```
program c2;
type address    = integer;
     byte       = 0..255;
     nibble     = 0..15;

  procedure doio(deviceNum: nibble; command: integer);

  { synchronous device driver  }

  const deviceAddr = -16833;  {337077 octal}
```

```
    type deviceWord = packed record
                        switch:  0..1;       {1 bit }
                        control: 0..7;       {3 bits}
                        device:  nibble;     {4 bits}
                        unused:  byte;
                      end;

         trick = record case boolean of
                   true:  (addr: address);
                   false: (wordptr: ↑deviceWord);
                 end;

    var  access: trick
    begin
      access.addr := deviceAddr;
      with access.wordptr↑ do
        begin
          switch := 1;
          device := deviceNum;
          control := command;
          repeat until switch = 0;
        end;
    end {doio};

  begin
     . . .
  end {c2}.
```

In this example, the record variant overlays an integer and a pointer to a packed record (both the integer and the pointer are word quantities). The integer variant is used to assign a memory address into the pointer. The pointer variant is used to access the fields of the record.

Bit field allocation in a packed record starts from the least significant bit of a word. Defining bit 0 as the least significant bit of a word and bit 15 as the most significant, the record declared in the previous example is allocated in the following fashion: `switch` is in bit 0, `control` occupies bits 1-3, `device` occupies bits 4-7, and `unused` occupies bits 8-15.

6.1.3 Bits

Bits are accessed in a similar fashion to bit fields but sets are used instead of packed records. The pointer variant is defined to point to a one-word set and set operations are used to test and set individual bits in the word.

To understand the following example it is important to be aware of the internal representation of a set. A set variable with a base type of 0..15 is represented as a word, with one bit corresponding to each value of the range. Bit 0 of the set variable corresponds to the first value of the base type. Successive base type values correspond to succesive bits of the set variable. When a particular value is present in the set, the bit corresponding to the value is 1. When the base type value is not present in the set, the corresponding bit is 0.

Example of arbitrary bit access:

```
unit bitter;
interface
  type address = integer;
       bits    = 0..15;

  procedure  SetBit   (addr: address; bit: bits);
  procedure  ClearBit (addr: address; bit: bits);
  function   BitIsSet (addr: address; bit: bits): boolean;
  procedure  FlipBit  (addr: address; bit: bits);

implementation

  type bitstring = set of bits;
       trick     = record case boolean of
                     true:  (addr: address);
                     false: (bitsPtr: ↑bitstring);
                   end;

  var access: trick;

  procedure SetBit {(addr: address; bit: bits)};
  begin
    access.addr := addr;
    access.bitsPtr↑ := access.bitsPtr↑ + [bit];
  end {SetBit};

  procedure ClearBit {(addr: address; bit: bits)};
  begin
    access.addr := addr;
    access.bitsPtr↑ := access.bitsPtr↑ - [bit];
  end {ClearBit};

  function BitIsSet {(addr: address; bit: bits): boolean};
  begin
    access.addr := addr;
    bitset := bit in access.bitsPtr↑;
  end {BitIsSet};

  procedure FlipBit {(addr: address; bit: bits)};
  begin
    access.addr := addr;
    if BitIsSet(addr, bit)
      then access.bitsPtr↑ := access.bitsPtr↑ - [bit]
      else access.bitsPtr↑ := access.bitsPtr↑ + [bit];
  end {FlipBit};

end {bitter}.
```

In this example, the record variant overlays an integer and a pointer to a set. The integer variant is used in each subroutine to assign the parameter `addr` to the pointer value. The pointer variant is used to access the set.

The parameter `bit` is used to designate which bit (0..15) is to be modified or examined. If `bit` is added to the set (set union) the bit corresponding to it (one of the bits 0-15 in the set) will be made 1. If `bit` is removed from the set (set difference) the bit corresponding to it will be made 0. The IN operation may be used to determine if the specific `bit` is currently in the set.

6.2 Unsigned Integer Manipulation

It is occasionally necessary to treat the contents of an integer variable as an unsigned (i.e., always positive) value. Integer operators expect signed values as arguments; thus, they must be used carefully when dealing with unsigned integers.

Integer variables are defined to contain values in the range -32768..32767, while unsigned integers are defined to contain values in the range 0..65535. Both representations are equivalent in the range 0..32767 but unsigned values in the range 32768..65535 are treated by integer operations as signed values in the corresponding range -32768..-1.

Integer arithmetic operators are +, -, *, DIV and MOD. Only +, - and * may be used with unsigned integers (as a consequence of their lack of overflow checking). DIV and MOD do not work correctly with large unsigned integers.

Integer comparison operators are =, <>, <, <=, > and >=. Only = and <> should be used with unsigned integers. The remaining operators do not work correctly with large unsigned integers. Unsigned comparison operators – as well as unsigned arithmetic operators – may be programmed by the user, as the following unit demonstrates.

Example of unsigned integer operations:

```
unit unsigned;

interface

    function uLess(a,b: integer): boolean;
    function uGtr(a,b: integer): boolean;
    function uLessOrEq(a,b: integer): boolean;
    function uGtrOrEq(a,b: integer): boolean;
    function uEq(a,b: integer): boolean;
    function uNotEq(a,b: integer): boolean;

    function uDiv(into, by: integer): integer;
    function uMod(into, by: integer): integer;
    function uPlus(a,b: integer): integer;
    function uMinus(a,b: integer): integer;
    function uMult(a,b: integer): integer;

    function uStr2Int(s: string): integer;
    procedure uInt2Str(int: integer; var s: string);
    function uInt2Real(int: integer): real;

  implementation

    function uLess {(a,b: integer): boolean};
    begin
      if (a >= 0) and (b < 0) then
        uLess := true
      else if (a < 0) and (b >= 0) then
        uLess := false
      else
        uLess := a < b;
    end {uLess};

    function uGtr {(a,b: integer): boolean};
    begin
      if (a >= 0) and (b < 0) then
        uGtr := false
      else if (a < 0) and (b >= 0) then
        uGtr := true
      else
        uGtr := a > b;
    end {uGtr};

    function uLessOrEq {(a,b: integer): boolean};
    begin
      uLessOrEq := not uGtr(a,b);
    end {uLessOrEq};

    function uGtrOrEq {(a,b: integer): boolean};
    begin
      uGtrOrEq := not uLess(a,b);
    end {uGtrOrEq};

    function uEq {(a,b: integer): boolean};
    begin
      uEq := a = b;
    end {uEq};

    function uNotEq {(a,b: integer): boolean};
    begin
      uNotEq := a <> b;
    end {uNotEq};

    function uDiv {(into, by: integer): integer};
    var i, OrigInto: integer;
        SeenPos    : boolean;
    begin
      if uGtr(by, into)
        then uDiv := 0
      else if into > 0
        then uDiv := into DIV by
  else begin
    i := 0;  OrigInto := into;  SeenPos := false;
    repeat
      into := into - by;
      i := i + 1;
      if not SeenPos
        then SeenPos := into > 0;
    until ((into = 0) or ((into < 0) and (SeenPos)));
    if into = 0
      then uDiv := i
      else uDiv := i - 1;
  end;
end {uDiv};

function uMod {(into, by: integer): integer};
begin
  uMod := into - (uDiv(into,by) * by);
end {uMod};

function uPlus {(a,b: integer): integer};
begin
  uPlus := a + b;
end {uPlus};

function uMinus {(a,b: integer): integer};
begin
  uMinus := a - b;
end {uMinus};

function uMult {(a,b: integer): integer};
begin
  uMult := a * b;
end {uMult};

function uStr2Int {(s: string): integer};
var int, i: integer;
begin
  int := 0;
  for i:= 1 to length(s) do begin
    int := (int * 10) + (ord(s[i]) - ord('0'));
  end;
  uStr2Int := int;
end {uStr2Int};

procedure uInt2Str {(int: integer; var s: string)};
var ones, tens   : integer;
    str1, str2   : string;
begin
  if int < 0
    then begin
      int  := int + 32767 + 1;
      tens := int DIV 10 + 3276;
      ones := int MOD 10 + 8;
      str((tens + ones DIV 10), str1);
      str((ones MOD 10),        str2);
      s := concat(str1, str2);
```

```
        end
        else begin
          str(int, str1);
          s := str1;
        end;
    end {uInt2Str};

    function uInt2Real {(int: integer): real};
    begin
      if int < 0
        then uInt2Real := 65536.0 + int
        else uInt2Real := int;
    end {uInt2Real};

  end {unsigned}.
```

This unit provides all comparison and arithmetic operators necessary for working with unsigned integers, as well as routines to convert unsigned integers to and from strings, and a function to convert an unsigned integer to a real.

NOTE: In Versions III and IV of the p-System, boolean comparisons generate unsigned comparison operators. Thus, the comparison functions can be programmed much more simply, as this example demonstrates:

```
  function uLess {(a,b: integer): boolean};
  begin
    uLess := ord(a) < ord(b);
  end {uLess};
```

Functions are provided even for operations that are identical to those for signed integers, in order to present a unified interface to the user. The code itself is relatively straightforward; `uLess` and `uGtr` are determined on the basis of sign, with negative numbers being larger than positive numbers. The other comparison operators are either identical to those for signed integers or built on the basis of `uLess` and `uGtr`.

Since signed integer divide does not work properly on unsigned integers, function `uDiv` simulates divide by repeated subtraction. Care must be taken to terminate the subtraction loop when `into` has fallen below zero; since negative values represent unsigned positive numbers, the variable `SeenPos` is used to determine when the loop has gone *past* all the signed positive values and has cycled back to the unsigned values. `uMod` is implemented using `uDiv`; this simple formula is adequate when only positive values are considered.

Input of unsigned integers can be accomplished using READ or READLN into an ordinary integer variable. Output can be performed by converting an unsigned integer to a string using procedure `uInt2Str`, then writing the value in the usual way. Procedure `uInt2Str` breaks the unsigned integer into two separate signed integer values, then converts these values to strings and concatenates them.

The function `uStr2Int` can be used to convert a string value to an unsigned integer variable. The function `uInt2Real` is useful when mixed unsigned integer and real calculations must be performed.

NOTE: Although for organizational purposes it is valuable to maintain the unsigned operators as interface procedures, calls to routines in units are very expensive relative to the operations performed within them. It is recommended that, in practice, simpler operations be coded inline.

WARNING: Consistent with the lack of integer overflow checks in UCSD Pascal, this unit does not perform overflow checks on unsigned integers.

Unsigned integer values are returned by the MEMAVAIL, VARNEW and TIME intrinsics. Section 6.1 describes how unsigned integers may be used as pointers.

6.3 Full-word Logical Operations

UCSD Pascal allows logical operations on word quantities (e.g., integers, characters and pointers). The standard functions ORD, ODD and CHR are defined as type transfer functions in UCSD Pascal. They do not modify the ordinal value of their arguments (see section 2.5). The boolean operators AND, OR and NOT are full-word operators. They do not mask off the high order bits of their result.

Example of full-word logical operations:

```
program logical;
var I: integer;
begin
  I := 556;
  I := ord(odd(I) and odd (255));
  { The high byte of I has been masked off }
  { I now contains the integer value 44    }
end {logical}.
```

6.4 Variable-sized Array Allocation

A "dynamic" or "conformant" array is an array whose bounds can be determined at runtime rather than at compilation time, as is ordinarily the case. The advantage of dynamic arrays is that the programmer may utilize only the memory necessary for a buffer, rather than declaring an array of the maximum possible size that may be required.

Although UCSD Pascal does not explicitly include the capability of declaring a dynamic array, the same effect may be produced using the system's dynamic variable allocation mechanisms.

The programmer first determines the increment by which the array may grow. In the examples that follow, this increment is a 512 byte block. The dynamic array will be an integral number of such blocks. A type declaration for an array of the desired type is included, of the precise size to occupy one such increment. Our examples contain a type declaration for an array of integers to occupy the block.

A variable-sized array is created in the heap by determining the dynamic bounds, sizing available memory, and coercing the system heap intrinsics to (a) allocate a memory space of the desired number of increments in the form of a dynamic array variable, given the size constraints, and (b) generate a pointer to that variable. The array is accessed as any dynamic variable (by de-referencing a pointer). Memory may be accessed beyond the declared size of the dynamic array variable (which was declared as containing only one increment) by indexing into the array with compile-time range checks suppressed (section 5.0.7). A program must maintain its own range checks by keeping track of the size of the allocated memory space and not indexing beyond its upper bound.

NOTE: The FILLCHAR, MOVELEFT and MOVERIGHT intrinsics are useful in manipulating variable-sized arrays. However, the byte count arguments to these intrinsics must be in the range 0..32767. Calls containing negative byte counts perform no action.

WARNING: Array indices are normally treated as signed integers. Indexing array elements greater than 32767 yields unexpected and often unfortunate results.

Buffer allocation strategies differ according to p-System version, since dynamic memory allocation is handled differently in Versions II and IV. Strategies are presented here for dynamic arrays using either the Version II heap intrinsics or the Version IV heap intrinsics.

6.4.1 Version II Heap Strategy

The Version II heap centers around the heap pointer. Dynamic variables are created by calls to the NEW intrinsic, which assigns the current value of the heap pointer to the associated pointer variable, and then moves the heap pointer up by the number of words allocated for the variable. The heap grows from lower addresses to higher addresses. As a result, consecutively NEWed dynamic variables are allocated contiguously in memory.

Programs may take advantage of this implementation feature to create variable-length arrays. A variable-length array is constructed by creating a series of fixed-size variables at runtime, and then treating them as a single, large array.

The UCSD Version II filer and editor use this method to create large arrays for manipulating file information. Their arrays are allocated in integral numbers of blocks. In order to create as large an array as possible, these programs use the MEMAVAIL intrinsic to determine the maximum buffer size allowed by the host system's configuration. MEMAVAIL returns the number of unused words in system memory; construction of a variable-length array consists of repeatedly NEWing one-block arrays until MEMAVAIL returns a value less than or equal to the program's memory threshold.

A memory threshold is defined as the minimum amount of memory required to execute a program (independent of the variable-length buffer). This includes space for variables and code segments used by the program in the course of its execution. The memory threshold for a given program is determined by making a rough estimate of the maximum space consumed by the program, and then tuning the estimate (usually through trial and error) to a minimum. A conservative minimum threshold for any program is 500 words (for system overhead) plus the program's requirements. If the program's memory threshold is estimated incorrectly and is smaller than necessary, the program will abort with a stack overflow error.

WARNING: On multitasking Version III implementations, this method of buffer allocation must be treated as a critical section (section 3.0) when tasks contend for heap space. The Version IV method (next section) is more secure in a multitasking environment.

Example of the Version II and Version III variable-sized buffer strategy:

```
program makeAbuffer;
const
     threshold = 1000;  {or whatever guess-timate}
     maxblks   = 60;
type
     block  = array [0..255] of integer;
     bufptr = ↑block;
var
     buffer, bufblock: bufptr;
```

```
      bufblks, index: integer;
  begin
    bufblks := 1;
    new(buffer);
    repeat
      new(bufblock);
      bufblks := bufblks + 1;
    until (bufblks >= maxblks) or
    ((memavail > 0) and (memavail <= threshold));

    { Note that bufblks * 256 < 32767 }
    { Array bounds = 0..(bufblks * 256 - 1) }

    fillchar(buffer↑,bufblks * 512,0);
    {$R-}
    . . .
    index := 756;
    buffer↑[index] := 4;  {or whatever}
    . . .
  end {makeAbuffer}.
```

Type `block` is declared as an array[0..255] of integer, which occupies exactly 512 bytes. The variables `buffer` and `bufblock` are pointers to type `block`. A NEW is performed on `buffer`, allocating storage for one block, or 256 array elements. The address of this pointer is saved in `buffer`. NEWs are repeatedly performed on `bufblock` (`buffer` cannot be used because the pointer to the original block allocated, which is the base of the array, must be preserved). New blocks are allocated until `maxblocks`, or the limit of available memory, is reached.

At this point, the integer variable `bufblocks`, which was incremented whenever a new block was allocated, reflects the total number of blocks allocated. Since each block contains 256 elements, there are a total of `bufblocks * 256` array elements, beginning at the location pointed to by `bufblock` and continuing in consecutive memory locations.

The FILLCHAR intrinsic illustrates how one may quickly and easily initialize the entire array to zero. No range checking is done with FILLCHAR, but since the array will be indexed with values greater than the declared upper bound of 255, range checking must be turned off using the compiler R– option.

6.4.2 Version IV Heap Strategy

When using the Version IV heap, successive NEWs are not guaranteed to allocate dynamic variables adjacently in memory. The VARNEW function is available for variable-sized buffer allocation. It accepts a buffer size request (in words) and allocates a buffer of that size. See section 3.5 for details.

The size of the largest possible memory buffer may be determined using the VARAVAIL function. It accepts a list of segments that might be memory-resident during the life of the variable-sized buffer. VARAVAIL returns the size of the largest memory space available, subject to the residency of all of the enumerated segments. The VARNEW intrinsic may be called to allocate a buffer of that size.

NOTE: Although the VARNEW intrinsic is protected from task contention, combination of the VARAVAIL and VARNEW functions may yield unexpected results in a multitasking environment. A "window in time" exists between a call to VARAVAIL and a call to VARNEW during which the memory reported by VARAVAIL may be allocated to another task. In this case, the call to VARNEW fails (VARNEW returns 0) and another call to VARAVAIL is required before the VARNEW may be retried.

Example of the IV.0 variable-sized buffer strategy:

```
program makeAbuffer;
const
     threshold = 1000;  {or whatever}
     maxblks   = 60;
type
     block  = array [0..255] of integer;
     bufptr = ↑block;
var
     buffer: bufptr;
     trash, bufsize, index: integer;
begin
  bufsize := varavail('makeAbuffer,fileops,pascalio') -
                                               threshold;
  if (bufsize < 0) or (bufsize > maxblks*256) then
    bufsize := maxblks*256
  else
    bufsize := bufsize - bufsize mod 256;
  trash := varnew(buffer, bufsize);

  { array bounds = 0..(bufsize - 1) }
  fillchar(buffer↑,bufsize*2,0);
  {$R-}
  . . .
  index := 467;
  buffer↑[index] := 4;
  . . .
end {makeAbuffer}.
```

Under Version IV, the available memory space is determined using the VARAVAIL intrinsic. The segments specified should include the program itself, any additional segments (including unit segments) it will be using during the life of the to-be-allocated buffer, and any resident operating system segments. VARAVAIL returns the available memory space in words. The program then scales this value down to the maximum buffer size desired, and to an integral number of blocks.

VARNEW is employed to allocate the entire dynamic array at once. The array is pointed to by the variable `buffer`. (In a multitasking environment, the variable `trash` should be checked to assure a non-zero value, as per the NOTE above.) As in the previous example, FILLCHAR is used to initialize the array (but this time `bufsize`, a word measure, must be doubled to reflect a byte count). The array is indexed as before, with range checking turned off.

VARDISPOSE may be used to deallocate the dynamic array.

6.5 Segment and External Procedures in Unit Interfaces

An implementation restriction in UCSD Pascal prohibits the inclusion of segment procedure headings in unit interface sections. This limits the routines accessible directly by the host to non-segmented procedures. Since it is quite natural to segment a unit according to routines as called by the host, a work-around must be employed; this is to simply establish "dummy" non-segmented procedures which do nothing but call the segment procedures that do all the work. The segment procedures themselves reside in the implementation section and are inaccessible to the host, but the "dummy" procedures may be in the interface section.

```
unit bigprocs;
interface
  procedure calproc1(myX: integer; var myY: integer);
  function  calproc2(myA: real): boolean;

implementation
  segment procedure proc1(x: integer; var y: integer);
  begin
    ...
  end;

  segment function proc2(a: real): boolean;
  begin
    ...
    proc2 := ...
  end;

  procedure calproc1;
  begin
    proc1(myX, myY);
  end;

  function calproc2;
  begin
    calproc2 := proc2(myA);
  end;
```

```
end.
```

Note that the parameters of the dummy and the segment procedures should correspond exactly in number and type.

External procedure headings may also not appear in unit interface sections. Thus, the following construction is illegal:

```
unit BadOne;
interface
  procedure AssemCode(i: integer); external;
```

It is possible, however, to place the procedure heading in the interface section without flagging it as external. The procedure can then be flagged as external in the implementation section. The following example illustrates how this is accomplished:

```
unit GoodOne;
interface
  procedure AssemCode(i: integer);
implementation
  procedure AssemCode; external;
end.
```

6.6 Structured Parameters Using Pointers

One of the areas in which Pascal shines is in its provision of user-defined data types. With this feature a programmer may define the data structure best suited to the application at hand, and write functions and procedures to operate on variables of the defined type. The UCSD Pascal Unit extension makes this a particularly handy feature, since the newly defined types and the subroutines which operate upon them can be written once and used by many application programs. Using Units, the Pascal language may be effectively extended; moreover the implementation of the extension is hidden from the programmer, whose only access to the unit is via the interface section.

Unfortunately, this scheme has a glaring Achilles' heel: A Pascal function may only return a scalar or pointer type. A Pascal function may not return a structured type. Consider the case of a unit written to handle dates. The basic data structure of such a unit might resemble the following:

```
type
  DateForm = record
               Month: 1..12;
               Day  : 1..31;
               Year : 1700..2500;
             end;
```

A convenient subroutine to have available might be `FutureDate`, which would accept a date and an integer as parameters, and return the date that is a number of days from a given date. The most straightforward way to write `FutureDate` would be as a function. To use `FutureDate` the programmer would simply write something like

```
NewDate := FutureDate(Today, 120);
```

where `NewDate` and `Today` are of type `DateForm`. However, since Pascal does not permit functions to return structured types, `FutureDate` may not be written in this manner.

One may write `FutureDate` as a procedure, invoking it as

```
FutureDate(NewDate, Today, 120);
```

where the formal parameter corresponding to `NewDate` is a VAR parameter used to return the appropriate value. The problem here is that it is often desirable to use the value returned by `FutureDate` as an intermediate value; it is cumbersome to force it to be assigned to a variable.

For example, if our unit includes a routine called `PrintDate`, which formats a `DateForm` value for output, and the programmer desires to print the date 120 days from today, it is inconvenient to have to write

```
FutureDate(NewDate, Today, 120);
PrintDate(NewDate);
```

It is far more convenient to be able to simply write

```
PrintDate(FutureDate(Today, 120));
```

saving both a statement and a variable. Of course, this again presumes the ability to return a structured type from a function.

6.6.1 Technique

An oft-used solution to this dilemma is to declare `DateForm` as previously described, but to also declare

```
type DateType = ↑DateForm;
```

All date variables would be declared as being of type `DateType` and would in reality be pointers. The unit would manage storage allocation; the application would not have to be aware of this representation. A primary advantage of this approach is that a function *can* return a pointer. Thus, it is permissible to write

```
PrintDate(FutureDate(Today, 120));
```

since `FutureDate` returns a pointer to a date variable, and `PrintDate` expects a pointer as argument.

The date unit would contain a central procedure – called `NewDate`, let us say – whose job would be to allocate new `DateForm` variables, assign values to the date fields, and pass the pointers to these `DateForm` variables to all functions in the unit which must return a value of type `DateType`. These functions would calculate the values to return; they would call `NewDate`, supplying it with the values and receiving from it a pointer to a variable of type `DateForm` to which those values have been assigned.

```
type
  Mtype = 1..12;
  Dtype = 1..31;
  Ytype = 1700..2500;
function NewDate(M: Mtype; D: Dtype; Y: Ytype): DateType;
var datepointer: DateType;
begin
  new(datepointer);
  datepointer↑.Month := M;
  datepointer↑.Day   := D;
  datepointer↑.Year  := Y;
  NewDate  := datepointer;
end {NewDate};

function FutureDate(
              now: DateType; days: integer): DateType;
var
  FutureMonth: 1..12;
  FutureDay  : 1..31;
  FutureYear : 1700..2500;
begin
  FutureMonth := ...;
  FutureDay   := ...;
  FutureYear  := ...;
  FutureDate  :=
           NewDate(FutureMonth, FutureDay, FutureYear);
end {FutureDate};
```

This approach is workable, and provides the programmer a neat interface to the data if sufficient routines are provided to allow the programmer to "forget" that pointers are being utilized. For example the very simple function

```
function GimmeMonth(dt: DateType): Month;
begin
  GimmeMonth := dt↑.Month;
end {GimmeMonth};
```

provides a painless way of extracting the `Month` component of a `DateType` without forcing the application programmer to be aware of the pointer implementation.

6.6.2 Heap Management

The problem here is that programs making heavy use of the Date unit as discussed so far will die quite quickly of stack/heap overflow. `NewDate` may churn out quite a number of `DateForms` but none of these are ever disposed of.

When the Date unit functions return pointers for assignment to `DateType` variables, the pointers must survive. But many times these functions return pointers to `DateForms` which are only used to contain intermediate results. The function call to `FutureDate`

```
PrintDate(FutureDate(Today, 120));
```

allocates space on the heap for a variable of type `DateForm`. The variable is used by `PrintDate`, but is then ignored, and is in fact inaccessible. Yet it continues to occupy heap space.

One solution to this problem was proposed by M.B. Feldman (*Byte*, Nov. '81). He suggests adding a boolean field called `IsTemporary` to (what in our example is) `DateForm`. The function `NewDate`, from where all allocated variables emanate, always sets `IsTemporary` to TRUE. A new procedure is added to the `DateAssign` unit. This procedure is used whenever it is desired to assign a value of type `DateType` to a bona-fide variable. First, `DateAssign` performs the assignment: One of the parameters to `DateAssign` is a VAR parameter of type `DateType`, the other is a value parameter of type `DateType`. The VAR parameter is assigned the value of the value parameter. `DateAssign` first DISPOSEs of the previous value in the VAR parameter, since it will be getting a new value. It uses `NewDate` to gain heap space for the new value of the var parameter. The value parameter is assigned to the var parameter, and the assignment part of `DateAssign`'s job is done – but then `DateAssign` sets the new dynamic variable's `IsTemporary` field to FALSE.

In this manner, a function call such as

```
PrintDate(FutureDate(Today, 120));
```

leaves the value returned by `FutureDate` flagged as `IsTemporary = TRUE`, since it is being used by the `PrintDate` routine, not being `DateAssign`'ed to a `DateType` variable. But the result of the procedure call

```
DateAssign(Tomorrow, FutureDate(Today, 120));
```

is to cause the value returned by `FutureDate` (and assigned to the `DateType` variable `Tomorrow`) to be flagged as `IsTemporary = FALSE`.

How do the variables flagged as temporary actually get deleted? Feldman suggests that each routine that *accepts a value of type* `DateType` *as parameter* check `IsTemporary`, and DISPOSE of that variable if the value is TRUE. For example, `FutureDate` checks the `IsTemporary` field of its first parameter, and `PrintDate` checks the `IsTemporary` field of its single parameter. They DISPOSE of their parameters if `IsTemporary` is TRUE.

In the first of the cases above, the actual parameter to `FutureDate` comes from a `DateType` *variable*, so no DISPOSal takes place. The actual parameter `Today` has had its value assigned to it via `DateAssign`, so its `IsTemporary` field is FALSE. The actual parameter to `PrintDate`, however, is returned by the function `FutureDate`. It has never been assigned to a variable, so `IsTemporary` remains TRUE. It will be DISPOSEd of by `PrintDate`, as it should be since it serves no purpose after `PrintDate` returns.

Unfortunately, this approach also presents problems. Look again at the example above, where `PrintDate` is passed the returned value of `FutureDate`. Suppose that `PrintDate` itself found reason to call on another Date unit routine, Routine `X`, passing to it `FutureDate`'s returned value as a parameter. Routine `X` performs its function. Then it faithfully cleans up the garbage by examining the `IsTemporary` field of its parameter – `FutureDate`'s returned value as passed on to it by `PrintDate`. It finds the value to be TRUE and DISPOSEs of the value!

When Routine `X` returns control to `PrintDate`, `PrintDate` finds itself without a parameter. Of course, `PrintDate` may still need that value, but there was no way for Routine `X` to know that!

The (inelegant) solution is for `PrintDate` to note the current value of `IsTemporary`, then set it to FALSE prior to calling Routine `X`. Routine `X` leaves the parameter as is. After Routine `X` returns, `PrintDate` has to restore the previous value of `IsTemporary`. Obviously, this method is fraught with danger.

Yet another problem with passing pointers rather than data as parameters is that every parameter becomes a VAR parameter. The pointer itself is local to the procedure, but the data being pointed to is the *same* data the calling routine uses – and any changes to `Months`, `Days` or `Years` will be global.

Nevertheless, if sufficient caution is exercised, a scheme such as the one outlined can yield excellent results.

6.7 Passing "Untyped" Parameters

Although Pascal's rigorous enforcement of the typing rules makes for more reliable programs and helps to prevent errors, there are times when it is necessary to defeat those rules. Earlier sections discussed how one may use the ORD and CHR functions to perform type conversions; overlaying techinques have been described, and will be further described in section 6.8.

The focus of this section is on the restriction that the type of the actual parameter passed to a routine must match the type of the routine's formal parameter. Often, particularly in systems programming applications, it is desirable to write a routine which may be passed a parameter of any type.

Consider a Screen Generator utility, which typically consists of two sections. The first section allows the programmer to define a screen layout, complete with input item positions and characteristics. These layouts are then stored in a file. The second utility section would be linked to an application wherein it would recall the screen, prompt the user to enter the data, and validate the data as it is entered.

How would one write the routine in the second section which would be called by the application to return a value from the screen? Assume the application programmer wants a value for the data item `ITEM1`. The application knows the type of `ITEM1`. The utility routine knows, from the screen file, the characteristics of `ITEM1`. But this information is available only at runtime. When the routine is being written, how should the formal parameter corresponding to `ITEM1` be declared?

The application programmer would like to be able to write `GetItem(ITEM1)`. (`GetItem` is the name of the screen generator utility routine which returns values entered onto the screen.) But each time `GetItem` is called, it is expected to return a value of a different type. The INTEGER type may be appropriate for `ITEM1`, but the CHAR type may be appropriate for `ITEM2`. The application programmer expects to be able to write `GetItem(ITEM2)`, however. Obviously, `GetScreen`'s formal parameter cannot be declared as both an INTEGER and a CHAR.

A solution to this problem is to have the application program pass to `GetItem` the addresses of `ITEM1` or `ITEM2`, rather than the `ITEMs` themselves. Now, a Pascal VAR parameter can be used to pass an address rather than a value, but type checking is enforced for VAR parameters.

Instead, the application discovers the address of the parameter in integer form, and passes this integer to `GetItem`. In this way `GetItem` always receives an integer, regardless of the original type of the `ITEM`.

For this solution to be practical, the application must be able to easily discover the address of any variable, regardless of type, and treat it as an integer. Also, `GetItem` must have a convenient means of accessing an "untyped" variable given its address.

To solve the first problem the application might be made to allocate each `ITEM` dynamically, and use a variant record to overlay the pointer with an integer. This would work, but would impose too great a burden on the application programmer. It is preferable to allow the application to declare the `ITEMs` as ordinary variables.

Instead, the application will be supplied with a function called ADDR which accepts a variable of any type as parameter, and returns the address of that variable. Such a function is standard in many languages but is not difficult to implement under the p-System. It is impossible to write such a function in Pascal, since Pascal requires that the parameter to a function be of one specific type.

The p-System does permit assembler language routines to accept VAR parameters without types. One can therefore write the ADDR function in assembler language, permitting it to accept an untyped parameter. The parameter is made a VAR parameter so its *address*–which is what we want–is passed to the assembler language function. It is then a simple matter to return that address as the function value.

Here is what the ADDR function looks like, coded in Z80 assembler language. The code would look essentially the same for almost any other processor, since the instructions do nothing more than move words onto and off of the stack.

```
        .func   addr,1
    ;
    ;This function returns the address
    ;of its single VAR parameter.
    ;
        pop     HL   ;pop return address and save it
        pop     BC   ;pop address of the parameter and save it
        pop     DE   ;pop "junk word"
        push    BC   ;push address of parameter on top of stack
        jmp     (HL) ;branch back to calling routine
        .end
```

The calling convention for assembler language functions under the p-System is as follows. First, the interpreter pushes one word of ”junk” onto the system stack. Then the addresses of the VAR parameters are pushed onto the stack (there is only one such parameter in this example) and finally, the return address is pushed onto the stack.

After the function returns, the interpreter expects to find these values removed from the stack. It expects to find the function's returned value on top of the stack.

The address of the VAR parameter must be extracted, so the return address is popped and saved (to enable a return from the routine), the address of the parameter is popped - this is the value which must be returned - and the ”junk” is popped, thereby clearing the function's stack.

The address of the parameter is now placed on the top of the stack and the routine returns.

The ADDR function may be assembled and placed in the system library. The application must declare it as an external routine:

```
function ADDR (VAR anyname {: Untyped}): integer; EXTERNAL;
```

Any valid identifier may be used as the parameter name in the function declaration; this identifier need never be referenced. Note that no type is specified for an untyped parameter to an assembler language routine.

The application uses the `ADDR` function to discover the addresses of the `ITEM`s in which `GetItem` returns screen entries, as illustrated below.

```
program application;
USES ScreenUtil;

var
  ITEM1: packed array[1..10] of char;  {or whatever}

function ADDR (VAR anyname {: Untyped}): integer; EXTERNAL;

begin
  ...
  GetItem(ADDR(ITEM1));
  ...
end {application}.
```

It has been shown how the application program may pass an untyped parameter to the `GetItem` routine. But as mentioned earlier, a convenient means for `GetItem` to access the `ITEM` once it has the address is required.

First, how does `GetItem` know the the type of an `ITEM`? Typically, these values are stored in the file associated with the screen. Or possibly the function `GetItem` explicitly passes the type as a second parameter. The application would declare

```
type
  MyTypes   = (MPackedArray, MInteger, MReal);
```

(or such a declaration would be imported from `ScreenUtil`) and would invoke

```
GetItem(ADDR(ITEM1), MPackedArray);
```

for example.

`GetItem` would declare a type corresponding to each type assumable by an `ITEM`. `GetItem` would also declare a pointer to each such type in a variant record, so that the pointer may be overlaid with an integer. When invoked, `GetItem` would select the appropriate type (as determined by the screen file or second parameter) and assign the integer-form address to the pointer to that type. The pointer would then actually point to the variable in the application (`ITEM`) whose address had been passed to `GetItem`.

The data entered from the screen could then be accepted and validated, and merely assigned to the pointed-to location. The item would instantly be "transported" to `ITEM`.

Below is an outline of what `GetItem` might look like.

```
Procedure GetItem(ItemAddr: integer;
                  ItemType: MyTypes
                  {an enumeration of possible types});
type
  APackedArray: packed array[1..10] of char;
  MPkdPntr = record
     {Would be used for screen items of APackedArray type}
               case integer of
                 1: (IntegerGuise: integer);
                 2: (PointerGuise: ↑APackedArray);
             end;
  MIntPntr = record
     {Would be used for screen items of integer type}
               case integer of
                 1: (IntegerGuise): integer);
                 2: (PointerGuise): ↑integer
             end;
     {And so on. One such record for each possible type
      assumable by a screen entry item.}
var
  PkdITEM: MPkdPntr;
  IntITEM: MIntPntr;

begin
  ...
  if ItemType = MPackedArray
    then begin
      PkdITEM.IntegerGuise := ItemType;
      read(PkdITEM.PointerGuise↑);
      {This would place the entry directly into the ITEM}
      {Now validate the entry};
    end
  else if ItemType = MInteger
    then begin
      IntITEM.IntegerGuise := ItemType;
      read(IntITEM.PointerGuise↑);
      {This would place the entry directly into the ITEM}
      {Now validate the entry};
    end
  else
 {and so on, for each of the possible
  types for screen entries}
   ...
end {GetItem};
```

The only messiness involved is the overlaying of pointers to each valid screen type with integers. That happens only in the `ScreenUtil`, however, and is hidden from the application programmer.

NOTE: The linker must be invoked to link the `ADDR` assembler language function to the application.

NOTE: The PMACHINE intrinsic presents another (more opaque) way for an application to discover the address of a variable. See section 4.26.

6.8 Variant Record Buffer Overlay

The previous section described how an "untyped" parameter might be passed to a routine. It is often necessary, however, to go beyond treating an isolated data item as untyped. Consider a utility to sort a file. Different files have records with differing structures; and even for one particular file it may be desirable to key a sort on different fields on different occasions. At the time the utility is written, therefore, no commitment can be made as to the size of the records or to the number, position, size or type of the key fields.

Essentially, the sort utility reads in buffers-full of the file and treats the buffer as one large "untyped" datum. All the utility can know, as it fills its buffers, is that it has a sizable array of bytes. Particular information about the file format or the sort fields must be passed to the utility upon each invocation. The utility must use this information to break the buffer down into individual records, and break the records down into individual fields. The fields must then be distinguished by type so that the appropriate operations may be performed on them (the sort, for example, has to determine whether to perform an integer or a character comparison on a key field).

One possible approach to this sort of situation is described in this section.

Let the utility define an "abstract array" type as a packed array[0..0] of 0..255. This is an array of bytes with only a single element. Define a pointer to this type as a variant record, so that it may be treated either as a pointer or as an integer (see section 6.8). The "abstract array" is used to manipulate the buffer. The variant record pointer/integer enables the "abstract array" to overlay whatever memory locations contain the file buffer to be manipulated.

Use the VARAVAIL intrinsic to size memory (section 3.5) and use the VARNEW intrinsic to allocate an area of words of the largest available size and provide a pointer to the area. Convert the pointer to an integer address using the ORD function. Assign this address to the pointer/integer to the "abstract array". The individual bytes in the "abstract array" can now be accessed using an index. Since the array is declared with bounds [0..0] (necessary since it is not possible to know the actual size when writing the program) it must be accessed with range checking disabled.

Use the BLOCK I/O intrinsics (see section 3.3.4) to fill this buffer area. Record-directed I/O (GET and PUT) cannot be used since the utility was written without knowledge of a particular file's record format.

The buffer may be traversed record-by-record by starting at the beginning of the buffer (`abstract_array[0]`) and adding the record length to the index of the current record to determine the index of the next record. The record length must be available to the utility; it may be passed as a parameter to it, or stored externally on a special information file pertaining to the file being manipulated.

Extreme caution must be exercised as the end of the buffer is approached. Since range-checking is disabled, the utility has the responsibility of assuring that it does not overshoot the end of the buffer. Further, since records may span blocks, it is likely that the record at the end of the buffer is only a partial record. In this case the utility must move the partial record to the beginning of the buffer using MOVELEFT, and BLOCKREAD another buffer-full from the file, beginning the byte *after* the end of the partial record. This process continues until the entire file has been traversed.

NOTE: Assure that reads always occur onto word-aligned addresses.

It is necessary, as the utility processes each record, to focus on and manipulate individual fields. The utility must be informed of the type and size of each such field, and its displacement within the record. Armed with this information the utility may "map" the fields to their appropriate types.

This can best be accomplished by defining yet another variant record type in the utility, which is referred to as a "many faces" type. The record has a variant for each type the utility might need to manipulate in a record. For example, if the utility can be expected to encounter fields of type integer, real or string, the "many faces" record has three variants, one for each type.

A pointer to "many faces" is then declared – but, as usual, the pointer is overlaid with an integer so that "many faces" can be placed at any desired memory location.

Now, when the utility decides to focus on a particular field, it merely adds the displacement of that field within the record to the index into "abstract array" which points to the beginning of the record. The sum is the index into "abstract array" of the field itself. This value yields an address which can be assigned to the pointer to "many faces". "Many faces" now overlays the desired field; the utility need only refer to the variant corresponding to the type of the field, and the field may be manipulated as any data item of that type.

Of course, such machinations are inelegant, and are to be avoided whenever possible. But for many general-purpose utilities, which must manipulate files of any type, they are unavoidable.

6.9 Data Prompts

This section describes the implementation of interactive data prompts. UCSD Pascal provides some support for interactive data prompts; one example of this is the acceptance of backspace characters when reading integers and strings from the console. But UCSD Pascal does not have the ability to respond to invalid user input, such as illegal characters in an integer or illegal file names in a file specification, without aborting the program. The system responds to invalid data and file names with an execution error. Therefore, responsibility for detecting and responding to invalid input falls to the program itself. This can be accomplished in a variety of ways. One way is to suppress I/O checks and use the IORESULT intrinsic to implement user-proof error recovery. Since IORESULT is also set as a result of bad input to the terminal, one can re-prompt the user for a valid entry until IORESULT returns zero. Since IORESULT is set as a result of attempting to access a non-existent file, one can re-prompt the user for a valid file name until IORESULT returns zero.

Another approach to validating input data is to perform character input exclusively, and attempt to convert the characters to the appropriate data type as they are entered. This method provides greater flexibility in that different kinds of errors can be met with tailored error messages, and invalid data can be trapped as soon as the first illegal character is typed, rather than upon completion of the entire entry. The entry need not be restarted, but may be resumed from the point of the error. Of course, this approach requires additional care to implement and additional time and space during execution.

The following four sections present robust implementations for single character prompts, integer prompts, file prompts and real prompts. The single character prompt provides a simple example of how data may be validated as it is entered. The integer and file prompts exemplify the use of IORESULT to detect bad input. The real prompt is a more complex demonstration of data being validated as it is entered.

The file prompt uses the file system conventions described in chapter 7.

NOTE: Though not demonstrated in this section, the GOTOXY intrinsic is useful for constructing interactive screen displays. See section 3.11.4 for details.

NOTE: There are a number of commercially available UNITs which perform "bullet-proof" data prompting for various data types.

NOTE: The UNITCLEAR intrinsic may be used to flush the keyboard type-ahead buffer before issuing a critical data prompt. This ensures that the prompt receives an explicit response from the user (rather than soaking up whatever characters happen to be queued for input). Examples of this feature can be found in the filer K(runch and R(emove commands.

6.9.1 Character Prompts

Example of character prompt:

```
program p1;
var ch: char;
    done: boolean;
begin
  done := false;
  repeat
    write('Do you wish to continue? (Y/N) ');
    repeat
      read(keyboard,ch);
    until ch in ['n','N','y','Y'];
    writeln(ch);
    done := ch in ['y','Y'];
  until done;
end {p1}.
```

This example demonstrates secure input checking. The prompt indicates acceptable responses to the question. The non-echoing keyboard file is used to filter out invalid responses before they can reach the screen; only when a valid response is received is the input written to the console. Note that the prompt accepts both lower- and upper-case characters as valid responses.

6.9.2 Integer Prompts

Example of integer prompt:

```
program p2;
var int: integer;
    done: boolean;
begin
  done := false;
  repeat
    {$I-}
    repeat
      write('Type a number (0 exits) : ');
      readln(int);
    until ioresult = 0;
    {$I↑}
    writeln(' You typed: ',int);
    done := (int = 0);
  until done;
end {p2}.
```

This example demonstrates explicit, user-defined error recovery using the system IORESULT value. If I/O checks were enabled, READLN would cause an execution error, terminating the program whenever the input did not match the format defined for an integer (e.g., input containing alphabetic characters). In this example, the input prompt merely repeats itself if an invalid integer is entered.

6.9.3 File Prompt

Example of file prompts:

```
program p3;
var infile,outfile: file;
    filename, inname: string[30];
    result: integer;

  procedure addSuffix(var fname:  string;
                          suffix: string);
  begin
    if fname[length(fname)] = '.' then
      delete(fname,length(fname),1)
    else
      fname := concat(fname,suffix);
  end {addSuffix};

begin
  repeat
    write(input file (<cr> to escape): ');
    readln(filename);
    inname := filename;
    if length(filename) = 0 then exit(program);
    addSuffix(filename, '.TEXT');
    {$I-}
    reset(infile,filename);
    result := ioresult;
    {$I↑}
    if result <> 0 then
      writeln(' Cannot open ',filename);
  until result = 0;
  repeat
    write(output file (<cr> for same): ');
    readln(filename);
    if length(filename) = 0 then
      filename := inname;
    addSuffix(filename, '.CODE');
    {$I-}
    rewrite(outfile,filename);
```

```
      result := ioresult;
      {$I↑}
      if result <> 0 then
        writeln(' Cannot open ',filename);
    until result = 0;
  end {p3}.
```

This example again demonstrates explicit, user-defined error recovery using IORESULT. If I/O checks were enabled, RESET and REWRITE would cause an execution error whenever an invalid file name was entered, but in this example the prompts reappear after responding with an error message. Note the use of file name suffixes. This conforms to the file system's naming conventions for file name prompts (as described in chapter 7). The user is expected *not* to enter the suffix, which will be automatically appended to the file name by the `addSuffix` routine. Also note the presence of a standard escape response for the input prompt (typing a carriage return escapes the prompt), and a short-circuit on the output prompt (typing a carriage return uses the input file title in the output file name).

6.9.4 Real Prompts

Example of real prompt:

```
program p4;
var
  r: real;

function RealRdln: real;
type
  charset = set of char;
var
  ch: char;
  base, exponent: real;
  CharsWritten: integer;
  DoOver: boolean;

procedure backspace(count: integer);
var
  i: integer;
begin
  for i := 1 to count do write(chr(8));
  for i := 1 to count do write(' ');
  for i := 1 to count do write(chr(8));
end;

procedure bell;
begin
  write(chr(7));
end;
```

```
function GetChAndQuit(valid: charset): boolean;
begin
  repeat
    read(keyboard,ch);
    if eoln(keyboard)
           then begin
             GetChAndQuit := true;
             exit(GetChAndQuit);
           end;
  if not (ch in valid)
    then bell;
  until (ch in valid);
  GetChAndQuit := false;
end;

function Ch2Digit(ch: char): integer;
begin
  Ch2Digit := ord(ch) - ord('0');
end;

function GetNumber(valid: charset; var Del: boolean):
                                                    real;
var
  KillSign, Minus, GotDec, GettingBlanks: boolean;
  NumOfPlaces: integer;
  returns: real;
begin
  NumOfPlaces := 0;
  returns := 0.0;
  Del := false;
  Minus := false;
  KillSign := false;
  GotDec := false;
  GettingBlanks := true;
  repeat
    if GetChAndQuit(valid)
      then begin
        if Minus
          then returns := -returns;
        if chr(13) in valid
        {which indicates that there will be no exponent}
               then begin
                      RealRdLn := returns;
                      exit(RealRdLn);
                    end
               else begin
                      GetNumber := returns;
                      exit(GetNumber);
                    end;
      end;
    if ch = chr(8)
      then begin
        Del := true;
        exit(GetNumber);
      end;
```

```
case ch of
  '0','1','2','3','4','5','6','7','8','9':
            begin
              if not GotDec
                then returns :=
                     returns * 10.0 + Ch2Digit(ch)
                else begin
                       NumOfPlaces :=
                            succ(NumOfPlaces);
                       returns := returns +
                       (1.0 / pwroften(NumOfPlaces))
                            * Ch2Digit(ch);
                     end;
              KillSign := true;
              GettingBlanks := false;
              write(ch);
              CharsWritten := succ(CharsWritten);
            end;
  '-'     : begin
              GettingBlanks := false;
              if KillSign
                then bell
                else begin
                  Minus := true;
                  KillSign := true;
                  write(ch);
                  CharsWritten := succ(CharsWritten);
                end;
              end;
  '+'     : begin
              GettingBlanks := false;
              if KillSign
                then bell
                else begin
                       KillSign := true;
                       write(ch);
                       CharsWritten :=
                          succ(CharsWritten);
                     end;
            end;
  '.'     : begin
              GettingBlanks := false;
              KillSign := true;
              if GotDec
                then bell
                else begin
                       GotDec := true;
                       write(ch);
                       CharsWritten :=
                          succ(CharsWritten);
                     end;
            end;
  ' '     : if not GettingBlanks
              then bell
              else begin
                     write(ch);
```

```
                         CharsWritten :=
                              succ(CharsWritten);
                       end;
      'E', 'e': begin
                  write(ch);
                  CharsWritten := succ(CharsWritten);
                  if Minus
                    then GetNumber := -returns
                    else GetNumber := returns;
                end;
    end {case};
  until ch in ['E','e']
 {or unless we exited previously upon a <CR>};
end {GetNumber};

begin {RealRdLn}
  CharsWritten := 0;
  repeat
    repeat
      base :=
         GetNumber(
['0'..'9','-','+','.','E','e',' ',chr(8),chr(13)], DoOver);
      if DoOver
        then begin
          backspace(CharsWritten);
          CharsWritten := 0;
        end;
    until not DoOver;
    exponent :=
           GetNumber(['0'..'9','-','+',chr(8)], DoOver);
    if DoOver
      then begin
        backspace(CharsWritten);
        CharsWritten := 0;
      end;
  until not DoOver;
  if exponent > 0.0
    then RealRdLn := base * pwroften(round(exponent))
    else RealRdLn := base / pwroften(round(-exponent));
end {RealRdLn};

begin {p4}
  repeat
    write('Input your value: ');
    r := RealRdLn;
    writeln;  writeln(r);
  until r = 0.0;
end {p4}.
```

This example illustrates how input data may be validated on a character-by-character basis, allowing only permissible characters to be entered and permitting the user to continue an entry after an error has been made.

The `RealRdLn` function contained in program `p4` permits the user to enter a real value in any reasonable format. Leading blanks are permitted, and a decimal point is not required for integral values. It is not required that a digit be on both sides of the decimal point. Signs are optional, as is the E for scientific notation. This is a much more liberal format than is accepted when using READ to directly input real values.

The heart of `RealRdLn` is the function `GetNumber`, which is used to accept both the mantissa and the exponent (if any) of the real value. `GetNumber` accepts a set parameter containing valid entry characters for the value desired. These differ for the mantissa and the exponent; for example, a decimal point is not permitted in the exponent. `GetNumber` returns the quantity entered as a real value.

`GetNumber` terminates when a carriage return, backspace or (for the mantissa) the letter E is encountered. A carriage return is used to signal termination of input. If the mantissa is being entered when the carriage return is encountered then no exponent is processed and `RealRdLn` exits. If the exponent is being entered when the carriage return is encountered, `GetNumber` exits so that `RealRdLn` can merge the exponent with the mantissa.

If a backspace is encountered, `GetNumber` returns with the VAR parameter `Del` set to true. The characters entered to that point are forgotten and erased from the screen. There is no facility for single-character backspace. (It is left as an exercise for the reader. Have fun!)

`GetNumber` validates each character as it is entered, and will only accept characters contained in the permissible set. It will also assure that permissible characters are encountered in the appropriate sequence. Thus, the only blanks allowed are leading blanks, only one decimal point is permitted, and only in the mantissa, and only one sign (per mantissa and exponent) and only one E are accepted.

An invalid character is not echoed; it generates a beep. It is a simple matter to modify `RealRdLn` to display an explicit error message in addition to the beep.

`RealRdLn` performs no range checking since real size is p-System implementation-dependent.

6.10 Device Drivers

A device driver is a set of one or more routines which provide an interface between a program and a peripheral device. The program initiates a device operation by calling the device driver with parameters describing the desired operation. The device driver performs the actions necessary to perform the device operation, and notifies the program of the device status.

Under the p-System, device drivers are normally written in the machine language of the host machine and made part of the interpreter. If, however, a non-standard device is attached to the system, it may be preferable to write the device driver as a Pascal UNIT, to be USEd by programs accessing the device.

This section describes how to write drivers for Q-bus ((tm) Digital Equipment Corporation) compatible I/O devices. However, the concepts are generally applicable.

Section 6.10.1 describes the interface between programs and device drivers. Section 6.10.2 explains how to access devices in UCSD Pascal.

6.10.1 Driver Interface

A driver typically consists of a set of functions or procedures declared in a program or unit. I/O driver parameters usually include a device identifier, a source or destination address, and a data transfer length. Depending on the driver, any one of these parameters may be implicit in the driver's definition and may not need to be supplied by the calling routine.

Examples of driver interfaces:

```
TapeRead (1{device number}, buffer, 512{byte count});
LPWrite (buffer, SizeOf(buffer){byte count});
TapeRewind (2{device number});
```

The source or destination address is normally an area of memory corresponding to a variable declared in the program. Parameter type checking may prevent general use of a driver for reading or writing data of varying types, since the parameter must be declared as being of one particular type. It is often necessary, therefore, to defeat the compiler's type checking constraints when writing device drivers.

One method of overriding type-checking constraints is the use of variant records, as discussed in sections 6.7 and 6.8. This allows the source or destination parameter to accept arguments having different types. The following example assumes that the programmer wishes to read data from a device into two different types of variables. The driver parameter is declared with a type allowing both kinds of variables.

Example of multitype parameter:

```
type  VariantStructure =
        record
          case integer of
            0 : (FirstStructure : Type1);
            1 : (SecondStructure : Type2);
        end {of VariantStructure};

procedure Driver (Var Buffer : VariantStructure);
```

The variant part may be extended in the following manner to accommodate byte-oriented drivers:

Example of byte-oriented address parameter:

```
type ByteArray = packed array [0..0] of 0..255;
     VariantStructure = record
       case integer of
         0 : (FirstStructure : Type1);
         1 : (SecondStructure : Type2);
         2 : (MemoryImage : ByteArray);
       end {of VariantStructure};
```

The driver may access any byte in the buffer by indexing through the `MemoryImage` variant. Note that range checks (see section 5.0.7) must be suppressed in order to access arbitrary bytes without causing an execution error. The program must assure that the index does not extend beyond the end of the structure.

6.10.2 Device Access

On many processor architectures, devices are accessed through their **device registers**. The driver views these registers as the contents of specific memory addresses. The computer is configured so that an access to those locations is mapped to the device's interface hardware, rather than to main memory. Accessing these memory addresses causes a device register to be accessed.

The device registers generally consist of a number of bytes for each device. Some of these bytes are reserved for status information regarding the device. Other bytes are buffers which serve as conduits into or out of the device. Data is transmitted through these buffers.

The precise memory locations which serve as device registers vary from one configuration to another. Configuring a device for a specific set of locations is usually accomplished with jumpers or switches on the device controller.

Device status information is obtained by reading from a device status register. Writing to a device register issues device commands.

Device register access is accomplished in UCSD Pascal by assigning the memory address of the device register to a pointer variable. This can be accomplished by overlaying the pointer variable with an integer, as described in section 6.8. The device register is then accessed through the pointer. Since most devices have several device registers located in contiguous addresses, the pointer is usually declared to point to a multiword record describing the device registers. The record fields are declared so that they coincide with the various bit fields in the device registers (see sections 6.0 and 6.1 for details).

NOTE: The process of storing into a packed record field involves reading the entire word containing the record field, updating the record field, and then writing the modified word back to the record. It is not possible to read or modify selected bits of a packed record without affecting an entire word. Since writing to a device register implies the issuance of a device command, the programmer must beware of side effects caused by reading and/or writing of packed fields adjacent to the field being modified.

The following example presents a simple device driver for the DLV-11 (a bidirectional serial line whose device registers in this example start at 1FF70 hex). A pointer is initialized with the address of the device register block. The program reads characters from the receiver and echoes them to the transmitter.

Example of simple DLV11 device driver:

```
program DLV11Demo;
const DLV11Address = -144;      {FF70 hex}
type
 DLV11Rec =
  record
    RCsr : packed record  {receiver status}
             Unused : 0..31;   {unused bits}
             IntEnab: boolean; {interrupt enable}
             Ready : boolean;  {char received}
           end {of RCsr};
    RBuf : char;           {input data}
    XCsr : packed record  {xmitter status}
             Unused : 0..31;   {unused bits}
             IntEnab: boolean; {interrupt enable}
             Ready : boolean;  {xmitter empty}
           end {of XCsr};
    XBuf : char;           {output data}
  end {of DLV11Rec};
var DLV11 : record
              case integer of
                0 : (Ptr : ↑DLV11Rec);
                1 : (Value : integer);
            end {of DLV11};

begin
  DLV11.Value := DLV11Address;
  with DLV11.Ptr↑ do
    begin
      RCsr.IntEnab := False;
      XCsr.IntEnab := False;
      repeat
        repeat    {wait for a char to arrive}
        until RCsr.Ready;

        repeat    {wait for xmitter to become available}
        until XCsr.Ready;
        XBuf := RBuf;      {send character received}
      until false;
    end;
end.
```

Note that changing a device's status by modifying a device register field is accomplished with a simple assignment statement. Thus, to disable interrupts for the sender and receiver it is only necessary to set their `IntEnab` fields to false. Similarly, reading the data from the receiver requires nothing more than saving the value of `RBuf`; in this case in `XBuf`, which causes the value to be sent to the transmitter.

Specific operational details of I/O devices can be found in the hardware documentation provided by the device's manufacturer. This information contains device register descriptions and operational assumptions.

Some devices are capable of performing direct memory access (DMA) operations. These devices provide a device register which contains the memory address of the next buffer element on which an I/O operation is to take place. The device driver must determine the starting buffer address and supply it to the device before a DMA operation is initiated. The address may be obtained with the PMACHINE intrinsic (see section 3.10 for details), which returns the address in a temporary variable. Alternatively, the `ADDR` function described in section 6.7 may be used.

Certain processors (such as the Intel 8088) do not normally have memory-mapped I/O. Instead, I/O is performed by reading or writing a data byte to a given port, the number of which is determined by the device's hardware controller. Since the p-System makes no provision for access to I/O ports from programs, user-supplied drivers must access ports through assembly language routines. A read-port routine for the 8088 might look like:

```
        .relfunc pread,1
        ; function pread(port: integer): integer;

result  .equ    8
port    .equ    6
        mov     bp,sp             ; get stack addressing
        mov     dx,(bp+port)      ; get port number
        in      al,dx             ; get data
        xor     ah,ah
        mov     (bp+result),ax
        retl    2
        .end
```

NOTE: Device drivers may require protection (i.e., semaphores) from task contention.

6.11 Locating Execution Errors

This section describes how to locate the source of an execution error in a UCSD Pascal program. When the operating system detects an execution error, it halts the program and displays an error message on the screen. At this point, the user may exercise one of two options: aborting the program which caused the error by typing <space>, or continuing execution by typing <escape>.

If a <space> is typed the system is reinitialized. Depending on the nature of the error, the program may or may not be restartable with the U(ser Restart main menu option. In either case, data and unclosed files from the failed execution are lost.

If <escape> is typed the program resumes execution from the point immediately following the error. Depending on the nature of the error, the program may or may not function correctly from that point.

NOTE: The <escape> option does not work correctly on a number of pre-Version IV p-System implementations.

The error message should be noted before either <space> or <escape> is typed since the error text may be erased. The error message includes a brief description of the cause of the error, as well as a field resembling the following:

```
Seg PROSE    P #90, O #101 <space> continues
```

This field specifies the error location in terms of the code file structure. The "Seg" value indicates the name of the current code segment. The P, or "Proc" value indicates the current procedure within the segment. The O, or "Offset" value indicates the procedure-relative byte offset of the instruction which caused the error.

6.11.1 Using Compiled Listings

A compiled listing displays the segment number, procedure number and code offset of each line in the program (see section 5.0.1 for details regarding compiled listings). Finding the source of an execution error consists of hunting in the listing for the named segment and for the Pascal statement whose procedure and offset numbers as listed match those of the error message. Note that while the procedure numbers can be matched exactly, the code offset displayed in the error message usually falls between the code offsets displayed in the listing, since each Pascal statement typically generates more than one p-code instruction. The error location can be narrowed down to the line whose listed offset is the closest value less than or equal to the error offset.

NOTE: In some situations, the execution error displays segment or offset numbers which do not appear in a compiled listing of the program. An execution error in an unrecognized segment may indicate a system problem or an invalid access of a library segment. Use the Library utility to confirm the presence of the offending segment in the operating system and call your p-System vendor for assistance.

NOTE: If an execution error is traced to a used unit, a compiled listing of the unit must be obtained before the error can be traced any further. When strings or long integers are passed as parameters, execution errors can occur in the vicinity of the associated procedure call. See sections 3.4 and 3.6 for more information.

Having located the suspected source line, the execution error message should be sufficient to determine the cause of the error. "Value range error" indicates that the program tried to assign a value outside of the declared bounds of an array or subrange variable. "Integer overflow" is only generated by long integer operations; it cannot result from integer operations (as no overflow checks are performed on integers). "Divide by zero" is detectable in integer, long integer and real division. User I/O errors are generated either by an invalid input or by a file system error.

NOTE: Some p-System implementations use a different format for certain execution errors which do not supply "Proc" or "Offset" values.

6.11.2 Without Using Compiled Listings

It is possible to trace execution errors to the procedure level without the use of compiled listings; all that is required is knowledge of the program's overall structure (i.e., declaration order of procedures) and an understanding of the compiler's rules for assigning procedure numbers in a compiled program.

Procedures in a program or segment are assigned procedure numbers in the order in which their headings appear, starting at procedure number one. (Note that forward declarations count as headings.) Procedure number one in a segment or program is the outermost block of the segment or program. In both cases, the first local procedure declaration is assigned procedure number two, the next three, and so on. Note that procedure numbers are assigned independent of the lexical nesting of procedures within a segment.

These procedure assignment rules may be partially verified by examining the compiled listing printed in section 5.0.1.

6.11.3 Further Investigations

Locating the source of an execution error is often only the first step in finding program errors; it is often necessary to begin printing debug information (by inserting WRITELN statements into the incorrect program) in order to investigate values of suspected variables prior to the execution error. The conditional compilation facilities available in later versions of the compiler are useful here; the programmer does not have to edit the debug statements out of the listing when the program is fully debugged.

The symbolic debugger, where available, is also a useful tool for debugging purposes. However, some knowledge of the architecture of the p-machine is required. Consult your Users' Manual for information regarding this utility.

6.12 Programming with Units

This section demonstrates the value of the UCSD Pascal UNIT in the economical and reliable production of applications software. The major benefits of unit usage are derived from their ability to act as a foundation for the development of increasingly complex facilities, their ability to be separately and independently compiled, and their ability to contain both global and private code and data.

The following two sections demonstrate unit usage by showing how to develop a unit and how to take advantage of pre-existing units. Unit syntax and semantics are discussed in section 3.2.

6.12.1 Unit Development

Program development using units is faster and more reliable than traditional methods. Large sections of code normally included in a program may be separated into units where they are available simply by reference rather than by in-line compilation. Facilities provided by such units are also available to any other programs requiring them. This approach saves time during program compilation and allows a unit to be tested and maintained independently of the program. Since a single copy of a unit is shared among all client programs, bug fixes and performance optimizations applied to a unit are automatically available to all client programs. Provided the interface of a unit is not changed, the unit implementation may be modified or enhanced without the need to recompile client programs.

The first step in the development of a unit is the specification of its interface section. When possible, it is prudent to structure interface variables and procedures to provide generalized functionality rather than facilities specific to an individual program. Only those procedures and data structures which must be directly accessed by the host should be included in the interface section.

Once the unit interface has been specified, it is wise to consider how each component of the interface is to be tested. This results in a greater understanding of all details of the interface functions, and provides a foundation for the construction of test and validation suites.

Implementation of the unit is performed by coding the interface functions and writing initialization and termination code for the unit's global variables (initialization code within a unit is a feature of Version IV and Apple Pascal; termination code within a unit is a feature of Version IV Pascal). Note that pre-existing units may provide functions valuable in implementing the unit (see following section) since units may use other units.

A unit may be designed to address a suite of related but separate functions. It should be kept in mind when coding such a unit that not every function will be utilized by every host. Separate functions should be localized to separate segments in order to minimize the total job memory requirements.

If the p-System environment supports selective USES (Version IV.1 and later), thought should be given to which data structures in the interface section are needed by which segments. This information should be provided in the documentation for the unit so that a host may select only those data structures required for the segments of the unit it will be using. Otherwise, the entire unit interface section will become global to the host, and compile-time memory will be wasted.

Once the unit has been compiled, it may be installed in the library system and tested. Testing and validation suites should be developed to exercise each component of the unit interface. These suites may be used during initial unit debugging and as a debugging aid during unit maintenance. Note that the unit may be programmed and maintained as an in-line unit of the validation suite program. This arrangement facilitates the validation of the unit after updates since the validation suite is always recompiled with the unit.

NOTE: It is often expedient to install the unit in a user library (see your Users' Manual) rather than in the system library, until debugging is complete. This obviates the necessity of constantly reconstructing the system library.

6.12.2 Using Pre-existing Units

A variety of units have been developed by a number of vendors for use with the p-System, including units which perform complex system, programming and applications functions. They afford access to routines that might be impossible for most programmers to write (i.e., routines that require intimate knowledge of the system architecture) or routines that might be merely inconvenient to rewrite each time they are required. Many p-System vendors include some of the more popular units as standard parts of their distribution; for example, the standard Version IV.1 distribution includes units to perform Filer functions from within a program and a unit to control the display of error messages from an application. Many hardware vendors include units to address specific features of their equipment. For example, a machine that supports an IEEE instrument port might include a unit to access that port.

The example below demonstrates the use of the WILD and DIRINFO units which are included in the standard release of Version IV. The DIRINFO unit permits programs to access the system date, and access and modify file names and dates, as well as access directories and perform additional file management functions. The WILD unit permits wildcard specification of file names; it is used by the DIRINFO unit. More complete information regarding these units may be found in the Version IV Users' Manual.

Although programs may directly read directories and perform the aforementioned functions without the use of units, DIRINFO, WILD and similar units provide a tested and standardized interface to the file system.

Program `DateFiles` uses DIRINFO to obtain the date from the root volume. It permits the user to select any textfile or list of textfiles as specified by a wildcard and re-date them to the root volume date. The DIRINFO unit is used to find all matching textfiles. `DateFiles` then asks the user to verify the date change for each file. If verification is obtained the date of the file is changed, again using DIRINFO. If verification is not obtained or if DIRINFO has problems locating the file the date is left unchanged.

Note that the majority of the code serves to link the complex functions of the units together.

```
program DateFiles;
uses Wild, DirInfo;
const
    Esc = 27;
var Lines,
    OutUnit  : integer;
    S        : string;
    VolDate  : DDateRec;

  procedure GetFileName (var Name: string);
  begin
    writeln;
    write ('File to re-date (you may use wildcards)? ');
    {This is the standard name-specification algorithm}
    readln (Name);  writeln;
    if length (Name) <> 0 then
      if Name[length (Name)] = '.' then
        delete (Name, length (Name), 1)
      else
        Name := concat (Name, '.Text');
  end {GetFileName};

  procedure GetVolDate (var TheDate : DDateRec);
  var List : DListP;
      Heap : ↑integer;
  begin
    mark (Heap);
    if (DDirList ('*:', [DVol], List, False) = DOkay)
                                   and (List <> Nil)
      then TheDate := List↑.DDate
      else writeln('Cannot find root volume!');
    release (Heap);
  end {GetVolDate};

  procedure UpdtFiles (Name: string);
  var List : DListP;
      Ch   : char;
      Heap : ↑integer;
  begin
    mark (Heap);
    if DDirList (Name, [DText], List, False) = DOkay then
      while List <> Nil Do
        begin
          Name :=
            concat(List↑.DVolume, ':', List↑.DTitle);
          repeat
            write ('Change date of ', Name, '? ');
            read (Ch);
            if not eoln then writeln;
          until Ch in ['Y', 'y', 'N', 'n', ' ', Chr (Esc)];
          if Ch in ['Y', 'y'] then
            if DChangeDate(Name, VolDate, [DText]) = DOkay
              then writeln('   Date of ', Name, ' changed.')
              else writeln('   Error — Date of ',
                                  Name, ' unchanged.');
          if Ch = Chr (Esc) then List := Nil
          else List := List↑.DNextEntry;
        end {of while}
    else writeln ('No files found');
    release (Heap);
  end {UpdtFiles};

begin
  GetVolDate (VolDate);
  repeat
    GetFileName (S);
    if length (S) <> 0 then
      UpdtFiles (S);
  until length (S) = 0;
end {DateFiles}.
```

6.13 Using Native Code

6.13.1 Automated Native Code Generation

Compilers that operate under the p-System generate p-code, a pseudo-code which must be interpreted as it is executed. The p-code is interpreted by the Interpreter, a program which is written in the machine language of the host processor. This interpretation process provides the benefit of portability; the entire p-System operating system and any software written for it can be moved to a different processor merely by implementing an Interpreter for that processor.

The benefit of p-code, therefore, is software portability. An additional benefit is object code compactness. Programs in p-code form are generally smaller than the equivalent program in native machine language. The cost of these benefits is speed. P-code programs generally execute more slowly than their machine language counterparts due to the overhead imposed by the interpreter.

For most applications the practical difference in speed between p-code and native code is negligible. On microcomputers, applications tend to be I/O intensive; much of the time an application is waiting on user input from the keyboard. However, applications that are processor-intensive may run at intolerably slow speeds when in p-code form. Applications that perform heavy numerical analysis fall into this category. Additionally, sections of code that include device drivers often need the speed available only from native machine language.

Version IV p-System's are available with a utility called a Native Code Generator (NCG), which converts selected portions of an already-compiled program from p-code to the native machine language of the host processor. The Native Code Generator is currently available for most of the popular processors which support the p-System.

Programs processed by the NCG generally run faster than the equivalent program in p-code form. However, these code files are slightly larger than their p-code equivalents, and are no longer portable to machines using different processors.

Sections of code which are to be translated to native code must be flagged in the source program using the N+ and N- compiler directives (see section 5.0.13). These directives typically bracket the BEGIN/END of a process or procedure (or a number of processes or procedures), since native code generation is accomplished on a procedure-by-procedure basis.

The Pascal compiler generates an executable p-code file; the flagged sections, however, contain information the NCG can use to perform its conversion.

The NCG is a boon to software developers since they can flag those sections of code that might benefit from conversion and distribute their software in p-code form. The user can execute the program as-is; on faster processors the conversion to native code may not be necessary. Converting to native code is an option left to the user – the user with access to an NCG for the appropriate processor may decide to use it or may decide to leave the program in p-code form. Portability need not be sacrificed until the product is in the hands of the end user.

Not all p-code is translatable to native code. Certain p-code sequences rely on interpreter-resident runtime support subroutines; these would translate to excessively large amounts of native code and are thus left intact. For example, the procedure and function calling sequences are very complex and are therefore not translated to native code. Thus, the major effect of native code generation on some sequences of code may be increased code size, with no substantial speed increase. Native code generation is best applied to variable accesses, computational loops, array indexing and other fundamental operations.

The interpreter scans for the soft <break> key and p-machine interrupts between execution of p-code instructions. During execution of a native code sequence the soft <break> key has no effect; a user's attempt to abort a program using <break> will fail during a native code sequence. However, one may force the interpreter to intervene during (what would be) a long section of native code by including calls to a dummy procedure. Procedure calls cannot be translated to native code so the interpreter is forced into play, and can detect a <break>.

NOTE: Version IV of the p-System also includes a utility called REALCONV. This utility converts real constants imbedded in a code file into their native machine language representation. Segments containing real constants will load and execute faster after being run through REALCONV. The REALCONV utility renders a code file non-transportable to a machine using a different real format.

6.13.2 User-Supplied Native Code

The UCSD p-System is available with assemblers for most major processors. Thus, native machine language routines may be used in conjunction with Pascal programs.

Details regarding effective use of the assemblers are beyond the scope of this book. Consult the appropriate assembler manual for this information. In this section a number of helpful hints and pitfalls to avoid when using the assemblers are briefly described.

Untyped parameters may be passed to assembler language routines. See section 6.7 for an example.

String constants should not be passed to assembler language routines. These are passed in a form that may be impossible to decode in the routine.

Assembler language routines delimited with .PROC and .FUNC headings occupy stack and heap space and cannot be moved or swapped. Routines with .RELPROC and .RELFUNC headings are in the codepool and can be swapped.

Routines delimited with .RELPROC should not allocate data areas using the .WORD, .BYTE or .BLOCK directives if the data in these areas must be preserved across calls to the routine. Since these routines may be swapped, the data is not preserved. Global data areas should be allocated using the .PUBLIC or .PRIVATE directives. Alternatively, the .WORD, .BYTE or .BLOCK directives may be used but the segment containing the routine must be MEMLOCKed to prevent swapping. Note that code segments residing in the code pool may be moved in memory by the operating system at any time. Pointers into such segments may be maintained across calls to the segment only if the segment contains .PROC or .FUNC routines and is therefore statically allocated in the stack/heap.

6.14 Passing Parameters Between Programs

The CHAIN intrinsic (see section 4.4) is a useful means of transferring control between one program and another. This ability is particularly useful when a system has a number of diverse applications integrated under the "umbrella" of a single master menu – the menu can CHAIN to any application, and each application can CHAIN back to the menu when finished.

A truly integrated software system not only permits the user to move from one application to another but also permits data to be transferred between the applications. For example, the results of a spreadsheet analysis might be transferred to a database for storage.

The CHAIN intrinsic does not provide for the general transmission of data from one program to another. The most straightforward means of accomplishing such a transfer is to write the data to an external file in the CHAINing program and to cause the CHAINed-to program to pick up the data from the file.

In situations where a significant volume of data must be transferred, using an external file may be the only practical solution. In many situations, however, only a few data items must be transferred; using an external file is a slow and cumbersome method of accomplishing the transfer under those circumstances.

A more practical scheme would be for the CHAINing program to deposit the data items in a set of specific memory locations and for the CHAINed-to program to pick up the data from those locations.

Such a scheme can be implemented as follows: A Data Unit (a Unit consisting of an INTERFACE section with data items) is written with a single data item – an array of as many bytes as is required to contain the data items to be transmitted from one program to another. This Unit is incorporated into the operating system (SYSTEM.PASCAL) using the Library utility.

Data segments in the operating system are allocated during the boot process. They endure through the transition between one program and another. Thus, a CHAINing program can USE the Data Unit and deposit the data it wished to transmit to the CHAINed-to program. The CHAINed-to program can USE the data unit and pick up the data.

The same Data Unit can be used for any number of CHAINing programs, even if they each transmit different kinds of data. The Data Unit should be established with an array of bytes large enough to accommodate the largest amount of data that might be transmitted. The CHAINing program can utilize the MOVELEFT intrinsic to copy the data to the Unit. The CHAINed-to program can utilize the MOVELEFT intrinsic to copy the data into an appropriately declared record.

A Data Unit:

```
unit common;
{compile this unit and install it in SYSTEM.PASCAL}
interface
type
  byte = 0..255;
var
  SharedData: packed array[0..99] of byte;
              {maximum of 100 bytes to be passed}
implementation
```

```
end.
```

A CHAINing program:

```
program ComingFrom;
uses commandio, {assumed to be in SYSTEM.LIBRARY}
                {$U COMMON.CODE} common;
var
  mydata: record
            item1: integer;
            item1: packed array[0..1] of char;
                                        {or whatever}
          end;
begin
  {body of program...}
  moveleft(mydata, SharedData, 4 {size of mydata});
  chain('GoingTo');
end.
```

A CHAINed-to program:

```
program GoingTo;
uses {$U COMMON.CODE} common;
var
  mydata: record
            item1: integer;
            item1: packed array[0..1] of char;
                                        {or whatever}
          end;
begin
  moveleft(SharedData, mydata, 4 {size of mydata});
  {... rest of program}
end.
```

NOTE: The scheme described in this section is analogous in its effect to the SETCVAL and GETCVAL intrinsics of Apple Pascal.

NOTE: Often, a suite of Pascal programs uses one or more common units. These units may be swapped out of memory as CHAINs occur from one program to another in the suite. Execution time will increase because of the need to reload these segments when the CHAINed-to program requires them. Installing the common units into SYSTEM.PASCAL may speed the execution of such program suites.

6.15 Coding Style and Optimizations

As with any compiler, the UCSD Pascal compiler produces more efficient code for some constructs than for others. This section describes the constructs for which the compiler produces smaller, more efficient code files. Use of these constructs may save as much as 30% of the overall code file size and execution time. Note that many of these constructs require the declaration of extra variables, which may reduce the amount of data space available. Care is advised when evaluating such tradeoffs.

6.15.1 Expressions and Array Indices

The UCSD Pascal compiler performs very little expression optimization. The compiler does not preevaluate expressions containing constants, nor does it reduce the complexity of array references. Therefore, programmers are advised to perform these optimizations themselves instead of depending on the compiler to perform them. The following program fragment is an example of three such situations:

```
var foon : integer;
    vreep: array [100..115] of integer;
    gnarl: array [0..5] of
             record
               fone : array [0..49] of integer;
               ftwo : integer;
             end;
begin
  <...>
  foon := 3 * (foon + 5) - 10;
  foon := vreep[foon + 4];
  foon := gnarl[3].ftwo;
  <...>
end;
```

The code generated by the UCSD Pascal compiler for the first expression includes one variable load, three constant loads and three integer operations. The programmer may algebraically reduce the expression to `3 * foon + 5`, saving one constant load and one integer operation, or 30 of the code.

The second expression involves an array with a 100-based index. The code generated by the compiler begins by loading the base address of the `vreep` array. It then calculates the array index `foon + 4` and subtracts 100 (the index base). Finally, it uses the result as an index off the array's base address to load the desired value. An optimizing compiler would have calculated the array index and indexed off of 100-less-than the array's base, saving the explicit subtraction of 100. A programmer may simulate this optimization by declaring all arrays with indices based at zero. (The UCSD Pascal compiler does not generate the needless subtraction of zero when array indices are based at zero.)

The third expression involves a constant index into an array of records and a load of the 51st word of that record. As in the second expression, the base address of the `gnarl` array is loaded. Next, the offset of the fourth record is calculated and added into the array base. Finally, the resulting record base is indexed to the 51st word, which is loaded and subsequently stored into foon. The p-System compiler generates special p-codes which combine the index and load operations when loading variables declared in the first eight words of a record. Hence, the second index operation could have been eliminated by rearranging the record to contain the `ftwo` field first. Unfortunately, nothing can be done to eliminate the first constant index. (Note that the compiler also generates a special index-and-store p-code for stores into variables declared as the first word of a record. Thus, the suggested rearrangement is doubly advantageous.)

The compiler generates range checking code for an array index (to make sure the calculated index is in the range of the declared index), any assignment to a variable declared as an enumerated type or any assignment to a variable declared as a subrange of a base type. Range checking may be eliminated at compile time using the Range Check compiler option (see section 5.0.7 for details). Elimination of range checks may reduce the size of a code file by up to 10, with a corresponding reduction in execution time.

6.15.2 Multiword Constants

In Version IV, code files contain a special section for multiword constants. The **constant pool** contains real constants, text literals and large set constants. There is a one-to-one correspondence between multiword constants in the source code and constants in the constant pool. Duplicate constants are neither detected nor eliminated. Hence, code space may be saved by assigning multiword constants to variables once, then using the variables thereafter.

This can produce particularly dramatic savings in the case of text literals, as the following example illustrates:

```
var
  s: string[18];
begin
  {wrong way}
  writeln('Please enter your name:             ');
  writeln('Please enter your address:          ');
  writeln('Please enter your telephone number: ');
  ...
```

```
{right way}
s := 'Please enter your ';
writeln(s,'name: ');
writeln(s,'address: ');
writeln(s,'telephone number: ');
...
```

When positioning text information on a screen, do not pad text literals with blanks to cause them to appear at particular column locations. Instead, use the SC_GOTOXY procedure from SCREENOPS (see section 8.1; pre-Version IV users should use the GOTOXY procedure; see section 4.12) to position the cursor, then write the string without the blanks, as below:

```
{wrong way}
writeln('                    Welcome to the XYZ System');

{right way}
sc_gotoxy(21,row);  writeln('Welcome to the XYX System');
```

Sets declared in the range 0..15 are represented in the code stream as individual p-codes. Sets declared in the range 0..4 are represented by a single byte. Sets declared in the range 0..7 are represented by two bytes and sets declared in the range 0..15 are represented by three bytes. Larger sets are represented as integral numbers of words in the constant pool, which require additional five-byte p-codes to access them. Bit for bit, it is more efficient to declare small sets instead of large sets when possible.

Set constants that contain a mixture of literal subranges and expressions are constructed at runtime, requiring relatively large amounts of code and execution time. For example:

```
sset := ['a'..'z', '0'..'9'];
tset := ['a'..'z', chr(13), '0'..'9'];
```

The `sset` is contained in an eight-word set constant contained in the constant pool. The `tset` is constructed at runtime from the eight-word set constant `['a'..'z']`, the calculated set value `[chr(13)]` and the four-word set constant `['0'..'9']`. When set constants containing expressions are used frequently, it is faster to assign the constant to a set variable and use the set variable instead.

6.15.3 Packed Field References

Declaring an array or record variable as a packed structure results in memory savings that vary according to the components of the variable. (The packing algorithms used in UCSD Pascal are described in section 6.0.) However, these savings are earned at the cost of larger code files and slower execution.

The compiler emits special p-codes for operations on packed structures. These p-codes construct and consume transient, multiword packedfield descriptors. Manipulation of these descriptors require between two and three times the code and execution time necessary to manipulate unpacked field descriptors. For example:

```
(*$R-*)
Program PCompare;
Const Bell = 7; (* Audible tone *)
Var I   : Integer;
    Arr : Array [0..0] Of Record
                            Fld2,
                            Fld1 : 0..15;
                          End;
    PArr: Array [0..0] Of Packed Record
                            Fld2,
                            Fld1 : 0..15;
                          End;
Begin
  Arr[0].Fld1 := 0;
  PArr[0].Fld1 := 0;
  Write ('Starting baseline test on <return>');
  Readln;
  For I := 0 To 30000 Do
    (* nothing, this is the baseline test *);
  Writeln (Chr (Bell), 'Stop timing');
  Write ('Starting unpacked test on <return>');
  Readln;
  For I := 0 To 30000 Do
    Arr[0].Fld1 := Arr[0].Fld1;
  Writeln (Chr (Bell), 'Stop timing');
  Write ('Starting packed test on <return>');
  Readln;
  For I := 0 To 30000 Do
    PArr[0].Fld1 := PArr[0].Fld1;
  Writeln (Chr (Bell), 'Stop timing');
End.
```

In the example above, the compiler generates 12 bytes of code for the unpacked array assignment and 16 bytes for the packed array assignment. On the IBM PC, the baseline loop executes in 9.1 seconds, the unpacked loop executes in 21.2 seconds, and the packed loop executes in 25.6 seconds. In this example, the packed array assignment required 1.3 times as much code as the unpacked version and took 1.4 times as long to execute.

6.15.4 Reals and Long Integers

Many applications require more precision than is provided by UCSD Pascal's 16-bit integers. Depending on the application, programmers may choose either real numbers or long integers to represent large values. Unfortunately, there is significant overhead involved in using either approach. In the case of real numbers, the p-code interpreter must be configured to provide real number operators and the operating system must contain real number I/O routines. In the case of long integers, the system library must contain the long integer library module. In either case, this code may add up to several thousand bytes, depending on the hardware facilities available. In addition, loading the real number I/O routines into memory may add seconds to the time required to start a program using reals; the long integer routines require a comparatively large amount of time to execute.

Certain applications requiring extended precision arithmetic may benefit from the use of explicitly programmed double integers. Double integer values are represented by two integers: one for the most significant part and another for the least significant part. User programs may manipulate these values in-line. By choosing an appropriate value for the maximum value of the least significant part, the precision of a double integer is maximized and the code necessary to manipulate it is minimized. For example:

```
Program Double;
Const Cutoff = Maxint;
Var LCount1, HCount1,
    LCount2, HCount2,
    Inner, Outer     : Integer;
Begin
  LCount1 := 0;  HCount1 := 0;  (* Set first value to 0 *)
  LCount2 := 0;  HCount2 := 0;  (* Set second value to 0 *)
  For Outer := 0 To 32000 Do
    For Inner := 0 To 32000 Do
      Begin
        LCount1 := LCount1 + 1;
        If LCount1 = Cutoff Then
          Begin
            HCount1 := HCount1 + 1;
            LCount1 := 0;
          End;
        If Inner = Outer Then
          Begin
            LCount2 := LCount2 + 1;
            If LCount2 = Cutoff Then
              Begin
                HCount2 := HCount2 + 1;
                LCount2 := 0;
              End;
          End;
      End;
    If (HCount1 > HCount2) Or
       ((HCount1 = HCount2) And (LCount1 > LCount2)) Then
      Writeln ('Count 1 is greater, by far')
    Else
      Writeln ('This isn''t supposed to happen');
  End.
```

In the example above, a double integer consists of a least significant part (with values between 0 and MAXINT-1) and a most significant part. The double integer has the value of the most significant part times MAXINT plus the least significant part. It is incremented by incrementing the least significant part. When the least significant part reaches MAXINT, the most significant part is incremented and the least significant part is reset to zero.

Although this method is somewhat inconvenient, the speed is comparable to using long integers. The double integers in this example have a range of approximately 0..2^30. (Greater capacity may be obtained using the unsigned integer techniques presented in section 6.2.) Unlike real numbers, no accuracy is lost due to rounding errors. Finally, double integers do not require the large support modules required by long integers and reals.

NOTE: Real constants embedded in p-System code files are represented in a machine-independent **canonical** form. Whenever a segment containing these canonical forms is loaded into memory, the operating system automatically converts all such real constants to the format appropriate for the interpreter and hardware being used. This process may take several seconds for segments containing a lot of real constants. This overhead may be reduced by assigning frequently used constants to variables and using the variables instead. The overhead may be eliminated entirely by using the REALCONV standard utility to convert all canonical reals in a segment to the native representation for a given machine. However, the price of this optimization is portability; the codefile that results is specific to those native floating point routines and cannot be executed on hardware that uses a different floating point format.

6.15.5 Short Forms

All versions of UCSD Pascal p-machines have p-codes that allow shorter and faster access to variables declared toward the beginning of a variable declaration section. Such p-codes are called **short forms.** Certain short forms apply to variable loads while others apply to variable stores.

The compiler generates short form p-codes using rules which depend on the **scope** of the variable being accessed. A variable's scope depends on its relationship to the code that accesses it. Variables declared in the procedure for which the code was generated are referred to as **local** variables. Variables declared at the program (or unit) level are referred to as **global** variables. Variables declared in any parent procedures (except the program itself) are referred to as **intermediate** variables, and variables declared in used units are referred to as **external** variables.

The table below indicates the variables for which the version IV compiler generates shorter and faster p-codes. The actual p-code emitted depends on a variable's **offset** from the beginning of the variable declaration section in which it appears. The offset of the first variable in a declaration is one; subsequent variables are assigned higher offsets. The lower a variable's offset, the shorter and faster the p-codes that access it. Section 5.0.1 describes how compiled listings may be used to determine the offset of a variable.

Variables beginning within the offsets specified in the table are accessed using p-codes of the specified length. Programs that declare their most frequently accessed variables at the beginning of their variable declarations are shorter and execute faster than their less careful counterparts by as much as 10. Note that accesses to local and global variables are the fastest types of accesses. Programs that access intermediate and external variables pay a premium in both execution time and in code space.

		Variable Loads		
		1-byte	2-bytes	3-bytes
Local	offsets	1–16	17–127	128+
*Intermediate	offsets		1–127	128+
Global	offsets	1–16	17–127	128+
External	offsets		1–127	128+

*One byte more is allocated for accesses to variables declared in the parent of a procedure's grandparent procedure or the procedures that contain it.

		Variable Stores			
		1-byte	2-bytes	3-bytes	4-bytes
Local	offsets	1–8	9–127	128+	
Intermediate	offsets			1–127	128+
Global	offsets		1–127	128+	
External	offsets		1–127	128+	

Note that these rules apply to the Version IV p-machine only. Pre-Version IV implementations have similar rules, except that all intermediate loads require between three and four bytes and the 1-byte local store operator is not available. An example illustrating the rules for Version IV is shown below:

```
Program Baklava;
Var Global1,                                     (* Offset 3 *)
    Global2,                                     (* Offset 2 *)
    Global3 : Integer;                           (* Offset 1 *)
    GArray  : Array [0..500] Of Integer;         (* Offset 4 *)
    GString : String;                            (* Offset 501 *)

  Procedure Parent1;
  Var IntStr1 : String;                          (* Offset 1 *)
      IntInt1 : Integer;                         (* Offset 42 *)

    Procedure Parent2;
    Var IntInt2 : Integer;                       (* Offset 1 *)
        IntStr2 : String;                        (* Offset 2 *)

      Procedure Local;
      Var Local1 : Integer;                      (* Offset 1 *)
          LString: String;                       (* Offset 2 *)
          Local2 : Integer;                      (* Offset 44 *)
          LArray : Array [0..5] Of Integer;      (* Offset 46 *)
      Begin
        (* From here, Local1 is local, all Intxxx's are
           intermediate, and all Gxxxx's are globals. *)
      End;

    Begin
      (* From here, all Intxxx2's are local, all Intxxx1's
         are intermediate, and all Gxxxx's are globals. *)
    End;

  Begin
    (* From here, all Intxxx1's are local, and all
       Gxxxx's are globals. *)
  End;

Begin
  (* From here, all Gxxxx's are globals. *)
End.
```

From procedure `Local`, the variable `Local1` is a local variable located within offsets 1 through 8. Thus, both loads and stores involving `Local1` use 1-byte p-codes and are hence very fast. Accesses from the `Local` procedure to the `IntStr1` and `IntInt1` variables use 2- and 3-byte p-codes respectively. Any access to the `GArray` variable uses a 1-byte p-code, even if a subsequent array index results in access beyond the 16th word. Note that the declaration order in the `Local` procedure is suboptimal; had the `Local2` variable been declared before `LString`, accesses to `Local2` would have used 1-byte p-codes instead of 2-byte pcodes. Note also that the compiler assigns offsets to `Global1`, `Global2` and `Global3` in the reverse order of their declaration. This phenomenum is called **reverse field allocation** and occurs whenever variables are declared of the same type by separating them with commas.

6.15.6 WITH Statements

The WITH statement is used to establish a temporary addressing environment within a procedure. A WITH statement allows the use of record field names without specifying the record name each time. As a side effect of this statement, the compiler calculates the base address of the record and stores it in a temporary. Subsequent access to the record's fields are performed by code that indexes off the address in the temporary instead of recalculating the record's base address and indexing. Using the temporary address as the record base saves the complex calculations normally required to access these fields. Programs that use the WITH statement are generally shorter, faster and easier to understand. For example:

```
Program WithDemo;
Type TheRec = Record
                 Field1,
                 Field2 : Integer;
                 Internal : Record
                              Field3,
                              Field4 : Integer;
                            End;
              End;
Var Beasty : Array [0..50, 0..20, 0..2] Of TheRec;
    Beauty : ↑TheRec;
    Tempty : TheRec;
    Field2,
    I, J   : Integer;
Begin
  Beasty[I * J, J * 2, I * 2].Field1  := 0;
  Beasty[I * J, J * 2, I * 2].Internal.Field3 := 99;
  With Beasty[I * J, J * 2, I * 2], Internal Do
    Begin
      Field1  := 0;
      Field3  := 99;
    End;
  Beauty↑.Field1  := 199;
  With Beauty↑ Do
    Begin
      Field1  := 199;
      Tempty.Internal  := Internal;
      Tempty.Field1  := Field1;
    End;
End.
```

In the example above, the `Beasty` array is a complex array of records which themselves contain records. Indexing to a particular array element involves the execution of a long sequence of code which consumes a lot of execution time. In the source code, the array indexing is so arduous as to obscure the meaning of the code. The first WITH statement evaluates the base address of the desired record. It also calculates the base address of the `Internal` record (using the the record base just evaluated). Subsequent record accesses use these base addresses instead of recalculating them. The resulting source code is much clearer and compiles into smaller, faster p-code.

The compiler may suppress the evaluation of a WITH record's base address if it is more economical to evaluate the base address as the record's fields are accessed. This may be the case when using a simple pointer in the WITH statement. Allocating a temporary to contain the base address and then indexing off the temporary yields no advantage over indexing off of the original pointer directly. Whether or not the compiler generates code to calculate the base address at the beginning of the WITH statement or calculates it inline, the WITH statement still establishes a local addressing environment within which only the field name is required to access a record's field. The WITH statement involving the `Beauty` pointer variable uses the value in `Beauty` as the base address of the object record instead of evaluating `Beauty↑`, assigning the result to a temporary and then indexing off the value in the temporary.

Note that the WITH statement involving the `Beauty` pointer variable contains assignments to `Tempty.Internal` and `Tempty.Field1`. Since corresponding fields of `Beauty↑` and `Tempty` have the same names, fields from `Tempty` must be fully qualified when used within this WITH statement; otherwise, the identically named fields in `Beauty↑` would be assumed. This causes the compiler to calculate the base address of `Tempty` twice within the `WITH Beauty↑` statement. In this situation it would have been more efficient to have used `WITH Tempty`, rather than `WITH Beauty↑`:

```
With Tempty Do
  Begin
    Beauty↑.Field1 := 199;
    Internal := Beauty↑.Internal;
    Field1 := Beauty↑.Field1;
  End;
```

The WITH statement above is equivalent in effect to the original, yet it generates more efficient code. This time, although the references to the fields of `Beauty↑` are still done directly off the pointer rather than off a temporary –as in the original code –the references to the fields of `Tempty` *are* done off a temporary –whereas in the original code the base address had to be recalculated for each reference.

WARNING: Since a WITH statement establishes a local addressing environment, ambiguity may arise between a field name declared within the record and identifiers accessible outside of the WITH. In the example above, it is anyone's guess whether the global variable `Field2` or the record field `Field2` would have been affected had there been an assignment to `Field2` in the first WITH statement. Bugs arising from such ambiguities are particularly difficult to locate.

WARNING: Values used in the calculation of a record's base address at the beginning of a WITH statement (such as array indices and pointer values) may be changed by code within the WITH statement without affecting the base address used throughout the WITH statement. However, should the compiler in some future release choose to evaluate the record's base address on each field access, this practice may yield code that produces unexpected or undesirable results. It is best not to change the values used to establish the local addressing environment until after the code contained within the WITH statement has executed. The following example illustrates what to avoid:

```
program avoid;
var
  point: ↑arry;
  arry: array[1..10] of record
                          thing1: integer;
                          thing2: char;
                        end;
begin
  new(point);
  with point↑.arry[inx] do
    begin
      thing1 := 10;
      inx := newvalue;  {Don't do this ...}
      new(point);
     {... or this, as the 10 and the 'N' may end up ...}
      thing2 := 'N';
     {... in different arry elements or different arry's!}
    end;
end.
```

6.15.7 String Manipulation

In general, the string intrinsics provide efficient operations on string variables. However, certain string intrinsics are grossly inefficient in certain circumstances. In these cases, use of more obscure methods results in smaller and faster code files.

The most inefficient string intrinsic is the CONCAT function. This function accepts a variable number of string values and returns the concatenation of all of the arguments (see section 4.6 for details). In order to implement this function, the compiler allocates a string temporary and initializes it to the empty string. Next, it calls the operating system string concatenation routine for each argument, accumulating the final function result in the string temporary. Therefore, the cost of a call to the CONCAT function is the allocation of a large string temporary and an operating system call for each string argument.

The INSERT intrinsic offers an efficient alternative to the CONCAT function. The INSERT intrinsic accepts a source string, a destination string and an index into the destination string as parameters (see section 4.15 for details). The CONCAT intrinsic may be simulated by inserting the first string argument into the second string argument. For example:

```
Program WinWin;
Var S1, S2, S3, Dest : String;
Begin
  <...>
  Dest := Concat (S1, S2, S3);
  <...>
  Dest := S3;
  Insert (S2, Dest, 1);
  Insert (S1, Dest, 1);
End.
```

In the example above, the CONCAT function results in the allocation of a string temporary, which is automatically initialized to the null string. A total of three calls to the operating system concatentation routine are made, in addition to the final assignment into the `Dest` variable. In the second half of the program, the CONCAT function is simulated using a string assignment and two calls to the operating system string insertion routine. The savings realized using this approach include a string temporary (41 words of local data space), the initialization of the string temporary and one call to the operating system.

Another inefficient string operation is the use of the string DELETE intrinsic to truncate a string variable. The DELETE intrinsic accepts a string variable, an index into the string and a character count as parameters (see section 4.8 for details). A call to the DELETE intrinsic involves the processing of the three parameters and a call to the operating system string deletion routine. A more efficient approach is to directly reduce the magnitude of the string's length byte. For example:

```
Program Expedient;
Var S : String;
Begin
  <...>
  Delete (S, Length (S) - 1, 2);
  <...>
  (*$R-*)
  S[0] := Chr (Ord (S[0]) - 2);
```

```
    (*$R↑*)
  End.
```

The example above demonstrates two methods of deleting the last two characters of a string. In the first instance, three parameters are processed and a call is made to the operating system string deletion routine. In the second instance, the string's length byte (which occupies the zero'th byte of the string) is decremented directly. Note that the length byte is accessed as a character variable, as if it were an element of the string. The CHR and ORD functions are used to allow arithmetic operations on the length byte. Additionally, range checking must be suppressed for the duration of the operation since access to the zero'th element of a string variable otherwise results in an execution error. The code generated for this approach is smaller and faster than for the approach using the DELETE intrinsic and does not require an operating system call.

6.15.8 CASE Statements

CASE statements are normally used when more than two possible actions may be performed based on the value of a scalar variable (otherwise an IF statement is used). CASE statements are implemented as a jump table; the compiler allocates one jump table entry for each value between the lowest value and highest value in the CASE statement. For example:

```
Program Bellicose;
Var I : Integer;
Begin
  <...>
  Case I Of
    -1000: Writeln ('Value is very small');
        0: Writeln ('Value, what value?');
     1000: Writeln ('Value is very large');
  End;
  <...>
  If I = -1000 Then
    Writeln ('Value is very small')
  Else If I = 0 Then
    Writeln ('Value, what value?')
  Else If I = 1000 Then
    Writeln ('Value is very large');
End.
```

In the example above, the lowest value in the CASE statement is -1000 and the highest value is 1000. Therefore, the compiler allocates a jump table containing 2001 words! The CASE statement may be simulated using the network of cascaded IF statements in the second half of the example. The compiler generates approximately three words for each IF statement (exclusive of the WRITELN call). In this example, the compiler generates approximately 100 times the amount of code for the CASE statement as for the IF statement network. Although the IF statement network executes more slowly than the equivalent CASE statement, in this case it is most cost effective to use the IF statements. In general, it is most cost effective to use a CASE statement when individual actions are provided for at least one third of the values between the lowest value and the highest value in the CASE statement.

6.15.9 GOTO Statements

One of the design goals of modern block-structured languages is to provide flow of control constructs that eliminate the need for the GOTO statement. Careless use of GOTOs may render a program unintelligible and error prone. However, Pascal does not provide constructs sufficient to eliminate GOTO usage entirely. Therefore, the GOTO statement lives on.

Since there is such a stigma associated with the use of the GOTO statement, programmers go to great lengths –sometimes inordinate lengths –to eliminate their usage. For example:

```
Program Boondoggle;

  Procedure ExitDemo;
  Begin
    <...>
    If SomeCondition Then
      Exit (ExitDemo);
    <...>
  End;

  Procedure GotoDemo;
  Label 1;
  Begin
    <...>
    If SameCondition Then
      Goto 1;
    <...>
  1:
  End;

Begin
  <...>
End.
```

In the example above, the `ExitDemo` procedure may be terminated by an Exit statement. The `GotoDemo` procedure performs the same action as the `ExitDemo` procedure, but uses the GOTO statement. The compiler generates approximately five bytes fewer for the `GotoDemo` procedure, and the GOTO statement executes far faster than the EXIT intrinsic. In this case, use of the GOTO statement is very defensible because its use results in more efficient code without loss of clarity or reliability.

6.15.10 Procedure Calls

In Version IV, certain types of procedure calls are faster than others and and certain types require less code. Analogous to the variable load and store operators described in section 6.15.5, the Pascal compiler emits procedure call operators that may call local, intermediate, global or external procedures. Some types of procedure calls have short forms. These short forms require fewer bytes of code, but execute only marginally faster than their more generalized forms.

The compiler generates procedure call p-codes based on the lexical level of the destination procedure relative to the caller. **Local** procedure call operators are generated for calls to procedures declared within the calling procedure. **Global** procedure call operators are generated for calls to procedures declared at the outer level of the program or unit containing the caller. **Intermediate** procedure call operators are generated for all other procedure calls to procedures within the segment containing the caller. **External** procedure call operators are generated for all calls to other segments (including used units). Since segment procedures may be declared either local, intermediate or global within a program or unit, external versions of the local, intermediate and global procedure call operators may also be generated. The external global procedure call operator is generated for all calls to used units.

The table below specifies the number of bytes emitted for each type of procedure call (exclusive of any parameter preparation):

Type of Call	Length
Local	2 bytes
Intermediate	3 bytes
Global	2 bytes
External Local	3 bytes
External Intermediate	4 bytes
External Global	3 bytes

Short forms of the external global procedure call operator are generated for calls to procedures in the first seven used units. In addition, short forms of the intermediate procedure call operator are generated for calls to a procedure's parent or grandparent procedure. Short forms require one byte fewer than the general form described in the table.

An example illustrating these rules is shown below:

```
Program Baklava;

  Procedure Parent1;

    Segment Procedure Parent2;

      Procedure Inner;
      Begin
    (* From here, Parent2 is external intermediate, Inner
       is intermediate, and Parent1 is external global. *)
      End;

    Begin
  (* From here, Inner is local, Parent2 is intermediate,
     and Parent1 is external global. *)
    End;

  Begin
(* From here, Parent2 is external local, and Parent1 is
   global. *)
  End;

Begin
  (* From here, all procedures are either global or
     external global. *)
End.
```

Note that in all versions of UCSD Pascal, external procedure calls are very slow relative to nonexternal procedure calls. When the segment containing the called procedure is not resident in memory, even the time necessary to execute the external call is small relative to the time required to load the missing segment from disk. However, when the required segment is already resident in memory, repeated calls to the segment have a detrimental effect on the time required to execute an application. Since calls to used units are implemented using external procedure calls, inter-unit calls are therefore discouraged (although execution within a unit procedes at the normal rate). The appropriate tradeoff between modular (unit-oriented) application construction and application speed must be determined on a case-by-case basis.

6.15.11 Parameters to Procedures

The Pascal language is constructed to encourage the passing of parameters between procedures. However, passing certain types of parameters by value require extraordinary amounts of time and space.

Strings, long integers, arrays and records are the most serious cases. Memory space for copies of such values is allocated within the dataspace of the called procedure; a parameter's value is automatically copied into its temporary memory as the called procedure begins execution. The larger the value parameter, the more temporary space is allocated and the more time is spent copying the parameter into this space. Some value parameters may be so large that there isn't enough space to allocate temporary memory. This overhead may be circumvented by passing the value parameter as a variable parameter. In this case, a pointer to the parameter is passed to the called procedure and no temporary memory is allocated. Note, however, that it is the responsibility of the called procedure not to modify the parameter (as it safely could if it was a value parameter) since such modifications are made to the actual parameter. For example:

```
Program YouPick;
Type Oink = Array [0..14999] Of Integer;
Var Pig : Oink;

  Procedure Fat (Death : Oink);
  Begin
    <...>
  End;

  Procedure Skinny (Var Careful : Oink);
  Begin
    <...>
  End;

Begin
  Fat (Pig);
  Skinny (Pig);
End.
```

In the example above, the `Fat` procedure accepts a 15000-word value parameter. A call to the `Fat` procedure results in the allocation of a temporary to contain a copy of this parameter. On many machines, there is not enough memory to allocate such a temporary, so such a call results in a fatal stack overflow error. A call to the `Skinny` procedure causes a pointer to the `Pig` array to be passed as the parameter. No temporary is allocated and it is the programmer's responsibility to assure that accesses to the `Careful` variable are read only.

In Version IV, passing a string constant as a value parameter is very slow relative to passing a string variable. This is because the code generated by the compiler in order to pass a string constant includes the LPR p-code, one of the slower operators in the Version IV p-machine. The code generated in order to pass a string variable does not include this operator. Therefore, passing string variables is much faster than passing string constants. Program execution time may be reduced by assigning string constants to string variables at program initialization time, then passing the string variables during program runtime.

Chapter 7

THE UCSD P-SYSTEM FILE SYSTEM

Contents

7.0 File System 244
7.0.1 General Overview 244
7.0.2 Syntax Overview 245
7.1 Physical Units 245
7.1.1 Syntax Overview 246
7.1.2 I/O Devices 246
7.1.2.1 Serial Devices 247
7.1.2.2 Block-structured Devices 247
7.2 Logical Volumes 247
7.2.1 Syntax Overview 248
7.2.2 Block-structured (Disk) Volumes 249
7.2.3 Disk Volume Usage 249
7.2.4 System Volumes 250
7.2.5 Prefixed Volumes 250
7.2.6 Disk Directories 251
7.2.6.1 Duplicate Directories 251
7.3 Disk Files 252
7.3.1 Syntax Overview 252
7.3.2 File Attributes 252
7.3.2.1 File Type 253
7.3.2.2 Data Files 254

7.3.2.3 System Restrictions Imposed by File Types 254
7.3.2.4 File Date 254
7.3.2.5 Size and Location Attributes 254
7.3.3 File Suffixes 254
7.3.4 File Titles 255
7.3.4.1 System File Titles 255
7.3.4.2 Other Reserved Titles 257
7.3.4.3 User File Titles 257
7.3.4.4 Titles with Non-block-structured Volumes 257
7.3.5 File Length and File Length Specifiers 258
7.4 Syntax Specification 259
7.5 Subsidiary Volumes 260
7.5.1 Creating and Initializing Subsidiary Volumes 260
7.5.2 Restrictions 261
7.6 File Conventions and Applications 262
7.6.1 File Name Prompt Conventions 262
7.6.2 Input Prompts 262
7.6.3 Output Prompts 263
7.6.4 File Access from User Programs 264

7.0 File System

7.0.1 General Overview

In the most abstract sense, a file is merely a sequence of data. A file system exists in order to adapt this abstract definition of a file to the requirements and constraints of a given hardware and software environment. The file system described herein has the following outstanding characteristics:

- Files may be accessed from Pascal programs with standard Pascal file operators.
- Files possess types ("extensions") to aid the user in identifying the contents of files and to increase system reliability by preventing invalid operations on files.
- The file system implements high level concepts such as removable disk volumes and device-independent file I/O.

- The disk file implementation is both time and space-efficient on relatively low-performance floppy disk drives, yet takes advantage of the additional speed and capacity of high-performance hard disk drives.

The following sections comprise a complete user-oriented specification of the file system. Section 7.0 presents an overview of file name syntax. Sections 7.1 through 7.4 describe the syntax and semantics of the file system hierarchy, starting with the lowest levels of device I/O and culminating in file attributes. Section 7.5 disucsses subsidiary volumes. Section 7.6 describes some system-wide conventions that apply to the file system.

References to file naming conventions and file system terminology throughout this book refer either implicitly or explicitly to the information presented in this section.

NOTE: In order to present a consistent file system description, this section defines a number of terms intended to describe parts of the file system. New terms are underlined and followed by either an immediate definition or a reference to a defining section. Subsequent occurrences of the defined term are not underlined.

7.0.2 Syntax Overview

```
<file designator>
---------------------------------- <file id> ------------->
          |                 ↑                         ↑
          |— <volume id> —|------------------------|
```

A valid **file designator** (informally referred to as **file name**) consists of a **volume identifier** and a **file identifier**. Volume identifiers and file identifiers are described in the following sections. The complete syntax for a file designator is presented in section 7.4.

7.1 Physical Units

Physical units correspond to I/O devices. They are addressed by pre-assigned physical unit numbers. I/O devices are defined to be either **serial devices** or **block-structured devices** (described in section 7.1.2.2). A **serial unit** is a physical unit assigned to a serial device such as a printer or terminal. A **block-structured unit** is a physical unit assigned to a block-structured device such as a disk unit.

The assignment of physical units to unit numbers is, to a great extent, implementation dependent. These assignments are typically implemented in the interpreter and are not user-modifiable. Exceptions to this rule include subsidiary volumes (described in section 7.5) and user-defined serial devices. Consult the Users' Manual for configuration details.

All configurations contain certain essential physical units. In addition, many configurations include specialized units, such as one which maps extended memory into a virtual disk (often known as a "ramdisk") or one which accesses the system clock. The table below shows physical unit number assignments typical for most implementations.

Unit Number	device description	unit attribute
1	screen and keyboard with echo	serial
2	screen and keyboard without echo	serial
4	disk drive 0	block–structured
5	disk drive 1	block–structured
6	printer	serial
7	remote port input	serial
8	remote port output	serial
9 – 12	disks 2 – 5	block–structured
13	first subsidiary vol	block–structured
nn	rest of " "	block–structured
nn + 1	user–defined serial	serial

7.1.1 Syntax Overview

Any physical unit may be used as a file. A file name corresponding to a physical unit may be constructed as follows:

```
<unit number>
————————————————  #<number>  ——————————>
```

The metasymbol <number> may be any valid physical unit number.

7.1.2 I/O Devices

I/O devices assumed to be connected to the system include disks, terminals, printers and remote ports. An I/O device has one of two states: online or offline. A device is online if it acknowledges status requests from the system and is available for I/O operations.

7.1.2.1 Serial Devices

A serial device either produces or consumes a byte-oriented sequence of data. Serial devices used with the system normally include terminals, printers and remote ports. The device drivers controlling a serial device assume that data transferred between the program and the device consists of human-readable data known as **text files**. In this context certain characters are treated as device control directives rather than as data. Alternate modes of serial I/O are available which make no assumptions about the data being transmitted. See sections 4.44 and 4.47 for details.

7.1.2.2 Block-structured Devices

A block-structured device is organized into a fixed number of 512-byte storage areas known as _blocks_. Blocks are randomly accessible by block number, starting with block 0. These devices are usually implemented as fixed or removable disks.

NOTE: High-capacity hard disks are often partitioned so that they appear to the p-System as a number of separate disk devices.

7.2 Logical Volumes

Logical volumes correspond to physical units; they are addressed by their assigned **volume name.** A **serial volume** is a logical volume assigned to a serial unit. A **block-structured volume** is a logical volume assigned to a block-structured unit. Serial volume name assignments are permanent and may not be changed by the user; serial volumes are functionally equivalent to their assigned serial units. Volume name assignments to block-structured units are dynamic and controlled by the user; a block-structured volume is addressable if and only if it resides on an online block-structured unit. Block-structured volumes are described in section 7.2.2.

All serial volumes may be used as files. Block-structured volumes should never be addressed as files except when using the file handler to create, examine and copy entire block-structured volumes.

Volume Name	Assigned Phys. Unit	volume attribute
CONSOLE:	1	serial
SYSTERM:	2	serial
<vol name>	4	block-structured

<vol name>	5	block-structured
PRINTER:	6	serial
REMIN:	7	serial
REMOUT:	8	serial
<vol names>	9 - 12	block-structured
<subsidiary vols>	13 - nn	block-structured
<user-def serial>	nn + 1	serial

NOTE: The volume number of the first subsidiary volume is user-configurable. See the Installation Guide for details.

NOTE: Volume names may be changed not only on removable-media block-structured devices, but also on fixed-media block-structured devices. Thus, the association of a volume name to a unit number for a Winchester disk may appear permanent as far as the user is concerned, since the disk will never be removed from the drive, but the system *does* allow the volume name to be dynamically changed.

7.2.1 Syntax Overview

```
<volume id>
------------ <unit number>: -------------->
       |                         ↑
       |---- <volume name>: ----|
       |                         |
       |----   *  --|-----------|
       |            v            ↑
       |---------------  :  ----|
```

The volume identifier may either be the **system volume** "*" (section 7.2.4), a unit number, or a volume name. File designators containing either empty volume identifiers or ":" specify the **prefixed volume**, which is described in section 7.2.5.

```
Examples:

CONSOLE:
SYS001:
#4:
*
*:
:
```

7.2.2 Block-structured (Disk) Volumes

Block-structured volumes (informally referred to as **disk volumes**) correspond to mass storage devices. The typical case is a floppy disk. A disk volume contains a collection of **disk files** (described in section 7.3). Information describing the files is centralized in a reserved area of the disk known as the **disk directory** (described in section 7.2.6). A disk directory also contains the volume name which identifies the disk volume. A disk volume is online if it resides on an online disk unit; it may be addressed by its volume name or by specifying the physical unit containing the disk volume, e.g., a disk volume named "SYSTEM" on unit 4 can be addressed either as "SYSTEM:" or "#4:".

NOTE: The disk volume associated with a floppy disk drive can be changed by inserting a new floppy disk in the drive. However, this may be a dangerous operation if there are open files on the volume.

NOTE: The volume name of an existing volume may be changed using the Filer C(hange command.

Details concerning the implementation of disk directories and disk files may be found in section 8.3.3.

7.2.3 Disk Volume Usage

Because disk volumes may be referenced by volume name, problems may arise when two disk volumes sharing the same volume name are online at the same time. This situation should be avoided whenever possible. When it is unavoidable, (e.g., a program makes an identical copy of a floppy diskette, including the directory) all file designators should avoid using volume names as volume identifiers. Instead, the physical unit numbers should be used to unambiguously specify files on online volumes.

When opening files on a disk volume the system searches all online block-structured units for the specified volume name. Because a floppy-based disk volume may not always be mounted in a particular floppy drive, disk volume names (instead of physical unit numbers) should always be used in conjunction with a file identifier specifying a disk file on the volume. Use of a physical unit to specify a volume would constrain the file system to search only the specified physical unit. The only exceptions occur when using the file handler to create, examine and copy entire disk volumes. Using a disk volume name as a file exposes the volume's disk directory to accidental overwriting by file write operations, thus threatening access to the volume's disk files.

WARNING: Removable volumes should never be switched when files are open on the volume. The operating system will not detect the switch and a write may destroy information on the switched volume.

7.2.4 System Volumes

The system volume, sometimes known as the root volume, is the disk volume from which the system was bootstrapped. It contains the operating system and usually the code files for the rest of the system parts. The system volume may be specified independently of its assigned volume name by using the volume identifiers "*" or "*:".

Normally, the root volume is unchanged from the time the system is bootstrapped until it is shut down. However, some manufacturers provide utilities which change the root volume from the bootstrap diskette to a volume on another block-structured device. When this occurs, necessary system files must be copied onto the new root volume. This procedure would be followed when a fast but volatile block structured device, such as a ramdisk, is available. The user would bootstrap the system from a diskette, then use the utilities to copy system files to the ramdisk and make it the root device. In such cases system response is greatly improved over the response obtained by using the diskette as the root volume.

7.2.5 Prefixed Volumes

The prefixed volume is used in conjunction with disk file designators. Normally, a disk file designator includes a volume identifier to indicate the volume on which the disk file resides in addition to the disk file identifier itself. Disk file designators lacking a volume identifier are assumed to reside on the prefixed volume. Thus, file naming can be simplified by specifying the most frequently accessed disk volume as the prefixed volume. The entire prefixed volume can be addressed with the file designator ":".

The default prefixed volume is the system volume. Another volume may be specified as the prefix volume in one of three ways: changing the system data structure which maintains the current prefix from within a program (not recommended), using the prefix redirection option, either when invoking a program or from within a program using the REDIRECT intrinsic (Version IV only; see section 4.29) or interactively, using the file handler P(refix command or redirection at X(ecute. If the volume identifier specified for new prefix volume matches the name of an online disk volume, the volume becomes the prefixed volume. The volume identifier can also specify an offline disk volume; when that volume comes online, it becomes the prefixed volume. If the volume identifier specifies a physical disk unit (as opposed to a volume name), whichever disk volume is mounted in the specified unit becomes the prefixed volume.

7.2.6 Disk Directories

Disk directories are stored on a disk volume along with disk files. Directories contain the volume name and up to 77 directory entries (but see section 7.5 on subsidiary volumes). A directory entry contains the name, location and attributes of a disk file on the volume. The file names in a directory must be unique in order to specify a file unambiguously; an existing file is automatically deleted if another file with the same name is entered in the directory. Disk file names are described in section 7.4. See section 8.3.3 for more detailed information on directory structure and contents.

NOTE: When the file system attempts to add a file to a volume containing a full directory, it prints the error message:

```
No room on vol
```

This is somewhat misleading, as the same message is used to indicate a lack of disk space.

7.2.6.1 Duplicate Directories

A disk volume may be marked so that the system maintains two disk directories on the volume. The second directory is called a duplicate directory and exists as a copy of the main directory. If unforeseen circumstances cause the destruction of the main directory, it can be restored using the information in the backup directory. The costs of duplicate directory usage are minimal: a slight increase in overhead due to the necessity of updating an extra disk directory during file manipulation and an extra four blocks on the disk to contain the duplicate directory. The insurance provided generally outweighs any losses in performance or space. A duplicate directory can be placed on a disk when it is initialized, using the file handler Z(ero command. The utility program Markdupdir may be used to create a duplicate directory on a volume at any time. The

Copydupdir and Recover utilities may be used to restore corrupted main directories (see your Users' Manual for details regarding these utilities).

7.3 Disk Files

Disk files are stored in an integral number of contiguous blocks on a disk and contain either programs or data. Each disk file has a number of file attributes, which provide useful information about the structure and history of a disk file. They are described in section 7.3.2. File names are the most important attribute of a disk file; they uniquely identify a disk file within a directory. File names are described in sections 7.3.3 and 7.3.4. File length directives control the amount of disk space allocated to a disk file. They are described in section 7.3.5.

7.3.1 Syntax Overview

```
 <file id>
 --------- <title> ------------------------------------->
                       |            ↑  |       ↑
                       |- <suffix> -|  |- [*] -|
                                       |       |
                                       |- [m] -|
                                       |       |
                                       |- []  -|
```

File **titles** distinguish the files in a directory. They are described in section 7.3.4. File **suffixes** allow the system and user to determine the contents of a disk file; they are closely related to file types. File suffixes are described in section 7.3.3. The syntactic items delimited by square brackets are **length specifiers.** Length specifiers serve as directives to the file system to determine the amount of disk space to allocate to a newly created disk file. They are described in section 7.3.5.

7.3.2 File Attributes

Disk file attributes are used by the system to manipulate the file and by the user to determine the contents and history of the file. From the user's point of view, the prominent file attributes are **file type** and **file date.** File types are described the following sections. File dates are described in section 7.3.2.4. The remaining file attributes visible to the user are **file length, starting block** and **bytes-in-last-block.** These are described in section 7.3.2.5.

7.3.2.1 File Type

All disk files have an attribute called the file type. File types enable both system and user to determine the contents of a disk file, regardless of its file name. **Text file** and **code file** are file types used by the system. Files of these types are described in section 7.3.3. Files containing subsidiary volumes are described in section 7.5. Files not containing text or code or subsidiary volumes are assigned the type **data file.** These are described in section 7.3.3. System restrictions imposed by file types are also described in section 7.3.3.

When a file is created, the system assigns a file type corresponding to the suffix. Subsequent file name changes do not affect the assigned file type.

The two file types described in this section are used to identify files containing specific internal structures. The structures are required (and assumed to be present and correct) by the system parts that operate on typed files. The internal structure of text files is described in section 8.0. Code files are discussed in detail in section 8.4.2.

Textfiles are usually created and maintained by the editor, although they can also be created by user programs. Text files contain human-readable text that represents either program source files, program data or written documents suitable for word processing. Serial devices used to display data for human scrutiny (e.g., consoles and printers) recognize text file conventions on output, thus text files written to serial units or volumes appear as they do in the editor.

NOTE: Text files have a specific structure; not every file containing text is a valid textfile, readable by the editor or by a program via READLN.

Code files are created by the compilers and assemblers, and are manipulated by the operating system and system utilities. Code files contain a mixture of p-code (possibly some native code) and execution information used by the interpreter and operating system.

Attempts to edit a code file with the editor or display a code file on the printer or console will fail; the system misinterprets the code file format as textfile information and spews forth a melange of audio/visual garbage for your entertainment (and possibly chagrin, since the garbage may contain values that destroy your terminal parameters). Code files are best examined and modified with the Patch and Decode utility programs. See your system documentation.

7.3.2.2 Data Files

Data files are created by programs and may have any internal representation. Except for being constrained to lie within an integral number of disk blocks, data files have no defined internal structure whatsoever. They match the Pascal language's definition of a file as a sequence of arbitrarily structured items.

7.3.2.3 System Restrictions Imposed by File Types

The editor does not accept files other than textfiles for editing. It uses the current suffix of a disk file name to guess its file type. This method of checking is sufficient for all practical purposes, however, it can be subverted by changing the suffix of an existing file name with the file handler.

7.3.2.4 File Date

The current system date is assigned to a file when it is created or modified (where "modified" is defined as the replacement of an old file by a new file of the same name or an update-and-LOCK of an existing file).

7.3.2.5 Size and Location Attributes

The length field indicates the number of blocks allocated to a disk file. The starting block field indicates the absolute block number of the first block of the disk file (block 0 is the first absolute disk block). The bytes-in-last-block field indicates the number of bytes in the last block of the file. This field is always set to 512 for text and code files, because they are created with block-oriented file operators; only data files have interesting values in this field. For data files, the system uses the bytes-in-last-block information to determine the end-of-file condition.

7.3.3 File Suffixes

File suffixes are separated from file titles by a period. File suffixes treated specially by the system are shown in the following table. A file created with one of these suffixes is assigned the corresponding file type. Otherwise, the file is designated a data file.

Suffix	File Type	System Uses
.TEXT	text file	text file identifier
.CODE	code file	code file identifier
.SVOL	subsidiary volume	
.BACK	text file	editor backup text file
.BAD	data file	damaged area of disk
.GRAF	vestigial file type; no longer applicable	
.INFO	"	
.FOTO	"	

.BACK files are backups of textfiles created by the advanced versions of the system editor. They share text file characteristics. .BAD files are created by the file handler X(amine command, and indicate a permanently disabled area of a disk. They are distinguished in that they are not moved when a volume is condensed and so will not be written over with good data.

7.3.4 File Titles

File titles uniquely identify disk files within a directory. The system reserves some titles for its own use. These are called system titles. All other valid file titles are user titles.

7.3.4.1 System File Titles

System files contain code and data used for system operation. They are identified by the file title "SYSTEM.<system part name>". The actual system files vary from implementation to implementation. Some of the more common system files with less than obvious functions are discussed below.

Most files with a code file type, except for the operating system (SYSTEM.PASCAL), are executable code files and can be invoked from the system prompt with the X(ecute command (if they are programs) or USED from within a program (if they are UNITs). System code files without a .CODE suffix (e.g., SYSTEM.FILER) may be invoked with the X(ecute command by following their names with a period "." (e.g., SYSTEM.FILER.).

NOTE: Most versions of the p-System perform certain preliminary setup operations when the compilers or assemblers are invoked from the main system promptline. Thus, the compilers and assemblers will not work properly when X(ecuted.

SYSTEM.MISCINFO contains information specific to a particular machine/terminal environment. It may be examined and modified with the Setup utility. Users may add their own routines to the SYSTEM.LIBRARY using the Library utility. These routines may then be used by any Pascal program.

SYSTEM.MENU (applicable only in Version IV) is the program which, when present, is automatically executed by the system at system bootstrap time. It effectively replaces the system prompt line.

SYSTEM.STARTUP is a user-defined program which, when present, is automatically executed by the system following the bootstrap, before displaying the welcome message or system prompt. It is used for turnkey applications programs which do not require other parts of the system. The difference between SYSTEM.STARTUP and SYSTEM.MENU is that SYSTEM.STARTUP is executed only when the system is initialized. SYSTEM.MENU is executed whenever the main promptline would appear.

While bootstrapping, the system searches for the interpreter and several system files on the system volume. These include SYSTEM.SPOOLER, SYSTEM.MISCINFO, SYSTEM.PASCAL, SYSTEM.MENU, SYSTEM.STARTUP and the work files. To locate the other system components, the system searches the system volume and then all other online disk units (ordered by increasing unit numbers) for a disk volume containing the system titles.

SYSTEM.SYNTAX is used by the compiler to generate error messages when a syntax error is discovered. If SYSTEM.SYNTAX is not present, only an error number will be made available.

Work files (SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE) exist to speed up interactive program development. The editor automatically reads in a the work textfile, if it exists and the compiler automatically attempts to compile the work textfile. The system R(un command automatically looks for the latest work code file to execute.

SYSTEM.SWAPDISK is used by the compiler on pre-Version IV implementations of the p-System (including Apple Pascal) to save memory during the compilation of large programs. If the following conditions hold:

- A 4-block file named SYSTEM.SWAPDISK resides on the same volume as SYSTEM.COMPILER.
- An "include" file directive is being processed, therefore, a disk directory must be read in order to open the "include" file.
- There is insufficient memory to read the directory, but the program's symbol table occupies more than 4k bytes.

then the operating system swaps a section of the symbol table out to the file SYSTEM.SWAPDISK, reads the directory into the resulting section of memory, opens the "include" file, and swaps the symbol table back into memory. The compiler will not automatically establish a SYSTEM.SWAPDISK; if it is needed but not present the compile will be aborted.

The program listing file optionally produced by the compilers and assemblers is named SYSTEM.LST.TEXT if no other name is specified.

The spool file, SYSTEM.SPOOLER, contains the names of files queued for printing when the spooler is active. Print spooling is available with Version IV.1 of the p-System.

The p-code interpreter, which must be available on all systems, is usually known as SYSTEM.INTERP or SYSTEM.<processor name> (e.g., SYSTEM.PDP-11). It appears in the directory as a datafile, and is written in the machine language of the host processor.

7.3.4.2 Other Reserved Titles

The file names <processor name>.OPCODES, <processor name>.ERRORS and USERLIB.TEXT are reserved for system use, in addition to the system file titles enumerated in the previous section. The OPCODES and ERRORS files are used by the Assemblers for opcodes and error messages, respectively. USERLIB.TEXT contains a list of user library file names. It is discussed in section 3.2.

7.3.4.3 User File Titles

User files may have any valid file title other than the reserved system file titles.

7.3.4.4 Titles with Non-block-structured Volumes

The file system allows the use of serial volume identifiers in conjunction with non-empty file titles (i.e., Console:.Text) even though serial volumes have no directories. In this case, the file title is ignored. This convention allows a system program to append a standard file suffix to a file prompt response without first having to determine whether or not the suffix is appropriate.

7.3.5 File Length and File Length Specifiers

When a disk file is created and made available for subsequent I/O operations, the file system must determine three things: whether the volume specified has an available directory entry for the new file, how much disk space to allocate for the new file, and whether the required disk space is available on the disk. When the I/O operations are complete, the system releases any disk space that was allocated to, but not used by the file. However, while the file is available for I/O, the system reserves all of its allocated disk space for growing room.

Files created without a length specifier are allocated the largest free space on the volume in order to minimize the possibility of growing files running out of disk space. This causes problems when a program attempts to create a number of new files on a disk volume having only one free space available. Although the number of blocks in the free space might easily contain all of the completed files, the first file created is allocated all of the available disk space, thus preventing the creation of other files.

File length specifiers change the file system's disk space allocation strategy in order to avoid problems such as the one described above. The value of the length specifier is treated as an estimate of the eventual maximum size (in blocks) of the file. The file system then allocates the specified amount of disk space for the file in the first free space large enough to contain it. For example, the file length specifier "[10]" allocates 10 blocks of disk space in the first 10-block chunk of free disk space.

The file length specifier "[*]" is useful when creating multiple files on a single disk; it allocates either half of the largest space on the disk or the second largest space, whichever is largest.

The file length specifiers "[0]" and "[]" are equivalent to a null length specifier. They allocate the largest space available.

If a growing file reaches the end of its initially allocated space, one of two things occurs. If the disk space immediately following the allocated space is already occupied by an existing file, the file system reports a system error. Otherwise, the space is part of a free space and the file's allocated disk size is extended to occupy the entire free space.

Length specifiers may appear in any file designator; however, they are ignored in any file operation other than file creation.

Free spaces are created on disk volumes as a consequence of normal disk file creation and destruction. Disk free space is managed with the file handler K(runch command.

7.4 Syntax Specification

```
<file designator>
-------------------------------------- <file id> ---------------->
             |                    ↑                        ↑
             |-- <volume id> --|------------------------|

<volume id>
--------------- #<number>: ---------------->
        |                          ↑
        |---- <volume name>: -----|
        |                          |
        |----   *   --|------------|
        |             v            ↑
        |------------------  :  --|

<file id>
----------- <title> ------------------------------------------->
                     |               ↑   |            ↑
                     |- <suffix> --|   |- [*] -|
                                         |          |
                                         |- [m] -|
                                         |          |
                                         |- [] --|
```

All spaces and control characters are ignored and all lower case alphabetic characters are mapped into their upper case equivalents. The following characters should not be used in a file designator: "$", "=", "?" and ",". These characters are treated specially by the file handler's file name prompts.

The volume identifier may specify a physical unit by its unit number ("#<number>:"), a logical volume by its volume name ("<vol name>:"), the system volume ("*:", "*"), or the prefixed volume (null, ":"). The volume name may contain any printable characters except "#" and ":", and has a maximum length of seven characters.

The file identifier consists of a title followed by an optional suffix and terminated by an optional length specifier. The title and suffix may contain any printable characters except "["; their combined maximum length is fifteen characters. A disk file's directory entry consists of the concatenation of title and suffix. This entry must be matched exactly by a file designator's title and suffix in order to locate the disk file.

The file length specifier is delimited by square brackets. The symbol "m" shown as one of the length specifier options denotes a positive integer.

Examples of valid file designators are:

```
*SYSTEM.WRK.CODE[*]
FOON.TEXT
SYSTEM.COMPILER
FLOPPY:SCRUB.BUB.FOTO[10]
:
```

```
*
*:
#12:
PRINTER:
DATA
```

7.5 Subsidiary Volumes

The p-System provides for a maximum of 77 files on a blocked volume. The p-System was originally implemented on machines with relatively low-capacity diskette drives as the sole means of secondary storage; it was rarely possible to store more than 77 files on a volume. However, this limitation is fairly extreme by modern standards. The 77 file limit would leave the volume with a large amount of unused (and unusable) space.

Version IV of the p-System introduced subsidiary volumes as a work-around to the maximum number of files per volume limitation. A subsidiary volume is a file on a block-oriented device which itself contains a directory and files. A subsidiary volume is both a file and a volume: it appears in the directory of the volume containing it (the principal volume) as a data file. However, it is can also be recognized by the file system as a volume in its own right.

7.5.1 Creating and Initializing Subsidiary Volumes

A subsidiary volume is distinguished from other files by its extension, which is .SVOL. A subsidiary volume is established using the file handler utility's M(ake command, specifying a file name ending in .SVOL. The file handler recognizes this extension; it establishes the subsidiary volume and initializes its directory. The user is asked if a duplicate directory is desired and is prompted for volume name.

It is recommended that the user specify a size, in blocks, when M(aking a subsidiary volume. Otherwise, the system assigns it the largest contiguous block of space on the specified volume, as when M(aking any file.

The maximum number of subsidiary volumes that may be online at one time is specified in SYSTEM.MISCINFO. If that number has not been reached at the time a subsidiary volume is established, the subsidiary volume is brought online (mounted) as soon as it comes into existence. When the system is bootstrapped, the online principal volumes are searched for subsidiary volumes; these are mounted as they are encountered, until the specified maximum is reached.

Offline subsidiary volumes are present on their principal volumes as files, but are not accessible as volumes. They may be mounted with the file handler's O(nline command. This command permits the user to dismount (take offline) currently online volumes (to allow for mounting other subsidiary volumes), to dismount all subsidiary volumes and to mount a specified subsidiary volume.

Subsidiary volumes may be discarded entirely using the Filer's R(emove command; the appropriate .SVOL files are purged from the volume directory and the subsidiary volume (with all the files it contains) is lost.

Subsidiary volumes may be mounted or dismounted from within a program using the DIRINFO unit, included in the Version IV.1 distribution.

7.5.2 Restrictions

Subsidiary volumes may not contain other subsidiary volumes. The system checks to assure that a volume is not a subsidiary volume before establishing a subsidiary volume within it.

The file handler's O(ffline command is picky about how subsidiary volume names are specified. When performing a M(ount, the .SVOL extension is *required*, though it is impossible to mount a file with any other extension. A volume name (ending with a colon ":") is not permitted when performing a mount. When performing a D(ismount, however, a volume name is *required*. A file name is not permitted, even a file name ending with .SVOL.

Note that in the file handler the M(ount and M(ake commands share the same one-letter invocation. To mount, the user must first have invoked the O(ffline command. However, it is easy to attempt to mount without first invoking O(ffline, and type the name of the subsidiary volume intended for mounting. The system interprets this as a M(ake command and attempts to establish a new subsidiary volume with the same name, destroying the old one! The user is urged to use the mount command with caution; if the system asks to remove the "old" subsidiary volume name, answer N(o!

If a principal volume containing an online subsidiary volume is taken offline and replaced with a different principal volume, the system is still under the impression that the subsidiary volume is online. Unit I/O to the subsidiary volume will destroy part of the new principal volume.

There is a distinction between the volume name and the file name of a subsidiary volume. When a subsidiary volume is established, its volume name (as written in its directory) is the same as its file name (ending in .SVOL) on the principal volume. However, either name may be changed using the appropriate file handler commands. Thus, a directory listing of the principal volume and the file handler V(olumes command may show different names for what may be the same subsidiary volume.

This is a potentially hazardous situation. Consider the following scenario: The user changes either the file name or volume name of an online subsidiary volume so that they no longer match. The user issues the file handler's R(emove command, wishing to permanently destroy the subsidiary volume. If the volume name were identical to the file name, the subsidiary volume would no longer be online; a sensible idea, since it no longer exists. However, with a different volume name, the system retains the subsidiary volume online. It permits the user to perform I/O to the subsidiary volume, thus destroying any new information stored in the space formerly occupied by the subsidiary volume.

It is recommended that subsidiary volume file and volume names never be made different.

7.6 File Conventions and Applications

This section describes some system-wide conventions for file name prompts. Programs developed by users should take advantage of these conventions in order to be consistent with the rest of the system.

7.6.1 File Name Prompt Conventions

File name prompts accept file names for one of two purposes: locating an existing file to use as an input file, or creating a new file to use as an output file. These operations are implemented with the UCSD Pascal file operators. See section 6.9 for details and examples.

7.6.2 Input Prompts

File prompts for input files appearing in the system are one of two kinds: type checking prompts and general prompts.

Type checking prompts enforce a weak form of file type checking by expecting only the volume identifier and file title for input. The suffix corresponding to the type of file expected (e.g., text file, in the editor) is appended by the system and the input file with the resulting file designator is opened. If no file with the appropriate suffix is found it is assumed that no file exists of the expected type. The operation is aborted. To handle situations where the desired file *is* of the appropriate type but does not have the corresponding suffix, type checking prompts provide a conventionalized "out": if the last character in the input is a period, a suffix is not appended (but the period is removed). For example, the X(ecute command expects a code file, and normally appends a suffix of .CODE to the specified file name. If a user supplies a file name of SYSTEM.EDITOR to the X(ecute command, the system looks for a file called SYSTEM.EDITOR.CODE, and aborts the X(ecute if no such file is found. However, the file name "SYSTEM.EDITOR." (Note the trailing ".") is accepted as a valid input file name identifying the file "SYSTEM.EDITOR". The period is stripped from the file name, but no attempt is made to append a .CODE suffix.

General prompts are more forgiving than type checking prompts. They accept any input as a valid file designator and proceed to open the file, taking the designator as stated. Only if the file system indicates that the file was not opened successfully will the proper suffix be appended to the input. The operation is retried with the appended suffix. A variation of general prompts is used by the compiler's "include" file mechanism (described in section 5.0.2).

7.6.3 Output Prompts

There are two types of output prompts appearing in the system, corresponding roughly to the type checking and general input prompts. The former variety is more reliable, and recommended.

Corresponding to the type checking prompt, the more reliable output prompts expect only the desired file title, concatenate the correct file suffix and create the output file. Examples of such prompts include the compiler code file prompt and the editor's output file prompt.

NOTE: These prompts do not check if a suffix is already supplied by the user, they merely append the appropriate suffix. Thus, if the user types "MY.CODE" in response to the compiler's code file prompt a file is created called "MY.CODE.CODE". This is a valid code file name, but is probably not the result expected by the user!

Corresponding to the general input prompt, an output prompt may accept any file specification and create the file with the name as literally stated. These prompts have a nasty habit of creating data files (instead of files with the expected type), because users accustomed to the former type of output prompt naively type only a file title as the output file name. An example of this sort of output file prompt can be found in most versions of the Library utility. The output file created by the Library utility is always a code file, but the utility does *not* automatically append a .CODE suffix to the user-supplied file specification. If a .CODE suffix is not explicitly supplied by the user, the Library utility will create a non-executable data file.

7.6.4 File Access from User Programs

All file system features and all file prompt conventions described in the previous sections are implemented with the same Pascal language features available to the user. No tricks are involved. This implies that user programs can take full advantage of the file system and prompt conventions for their own prompts.

Chapter 8

SYSTEM UNITS AND DATA STRUCTURES

Contents

8.0 TextFile Format 267
8.0.1 File Structure 268
8.0.2 Header 269
8.1 Using SCREENOPS Data Structures and Procedures 270
8.2 COMMANDIO Monitor and I/O Redirection 277
8.3 The KERNEL 280
8.3.1 System Constants 281
8.3.2 Accessing the System Date 281
8.3.3 Using Directories from Programs 282
8.3.3.1 Reading and Modifying Directories 283
8.3.3.2 Duplicate Directories 288
8.4 Segment Code Management 288
8.4.1 Introduction and Overview 289
8.4.2 Code File Structure 293
8.4.2.1 Segment Dictionary 294
8.4.2.2 Segment Format 297
8.4.2.3 Segment Reference List 300
8.4.3 Environment Records & Segment Information 301
8.4.4 As a Program Runs 305
8.5 File Information Blocks (FIBs) 306
8.6 Accessing Internal Operating System Procedures 309

8.7 The Compiler/Operating System Interface 319

In the best of all possible worlds, a Pascal programmer would never need to be concerned with internal operating system data structures or with low-level system implementation details. Unfortunately, neither the p-System, nor any other operating system, provides its users with the best of all possible worlds.

At times, it is necessary for an application to peek into hidden parts of the system to discover information that is otherwise not available. It may be necessary to discover terminal or hardware attributes that differ from installation to installation, for example. This information is contained in an operating system data structure. Or, an application may need to manipulate a disk directory in an unusual manner and so not be conveniently able to use the standard system interface to this structure.

The trend, happily, is to make these low-level constructs available in a convenient and controlled manner through the use of special-purpose units. Thus, later versions of the p-System come with units for screen control, directory access and a variety of other functions. Programs using these units benefit from tested and "standardized" code; the risk of possibly crashing the operating system is greatly minimized.

But situations will inevitably arise which will require low-level access not provided by canned units. And earlier p-System versions often provide the programmer with no recourse other than explicitly accessing system data structures.

A complete explication of the p-System operating system is beyond the scope of this book; in any case, a complete understanding of the operating system is rarely if ever necessary. This chapter discusses those operating system details a programmer will most likely need to know.

Section 8.0 describes the internal format of text files. Section 8.1 details SCREENOPS, a unit provided with Version IV of the p-System for performing screen I/O in a terminal-independent fashion. Section 8.2 mentions data structures and procedures contained in the COMMANDIO unit beyond those discussed in previous sections (see sections 4.4, 4.9 and 4.29). In section 8.3 the KERNEL, which contains most of the operating system data structures, is analyzed in detail. Section 8.4 discusses the structure of code files and their management by the operating system. The internal representation of files is the subject of section 8.5. Section 8.6 details how the programmer may access internal operating system procedures. Finally, section 8.7 covers the interface between the operating system and the various language translators it supports.

A word of caution: low-level implementation details are notoriously version dependent and are subject to change without notice (a good reason to use the provided units whenever possible!). Code that works under one p-System version may require modification before it works on a later version, or it may not work at all.

8.0 Text File Format

Ordinary p-System data files have no format beyond that imposed upon them by the data structures describing their records. They consist entirely of the data explicitly written to them, with no "filler" bytes and no extraneous records.

P-System text files, on the other hand, have a unique and specific format. They contain additional information besides the text itself, including control and filler characters. And they begin with a two-block "header" record that describes characteristics of a particular textfile.

Ordinarily, these text file features are transparent to the p-System programmer and user. A filer transfer to the console or printer appears as plain text. An edit session displays only text. A read or write from or to a text file from within a Pascal program need not take into account the additional non-textual material within the file.

The various parts of the operating system cooperate to hide the extraneous textfile information from the textfile user.

There are situations, however, when it is desirable to be aware of a text file's internal structure. For when a program places data into a text file using BLOCKWRITE or UNITWRITE the system no longer handles the details of the file's internal structure automatically. The operating system does not intervene when these intrinsics are employed, and the file contains precisely what is written to it.

Unfortunately, subsequent attempts to access that file as a text file are likely to fail, since the system will expect to find the specific text file format. Therefore, programs writing text files with BLOCKWRITE or UNITWRITE have the responsibility of explicitly handling the text file's internal structure.

Typically, this situation will arise when a program must write to the textfile as quickly as possible, so that it may return to a task that requires its urgent attention. As an example, consider a terminal emulator program which is receiving text from a remote source and logging it to a text file. The program will accumulate a buffer-full of text, then dispatch the text to the log file. The faster this operation takes place, the less likely that text arriving through the remote port will be lost.

Section 8.0.1 discusses the internal structure of a text file. Section 8.0.2 discusses the textfile header, a two-block record which appears at the beginning of every textfile.

8.0.1 File Structure

A textfile is composed of units called "pages", each of which is two blocks (1024 bytes) long. The first page is the header, which will be discussed in the next section. Succeeding pages contain the text itself in the form of ASCII characters.

All text files contain header records and a minimum of one page of text. Therefore no textfile is smaller than four p-System blocks in length. As the text file increases in size, pages are added to accommodate the additional text. Therefore a textfile must always have an even number of blocks.

Text file records are lines of text terminated by the ASCII carriage return character (a byte containing decimal 13). A page must always contain an integral number of lines; no line may be broken over more than one page. (Note that there is no restriction applying to lines crossing block boundaries.) Since pages are always uniform in size, there will often be bytes remaining following the final carriage return of a page. These bytes are filled with nulls (bytes containing zero). Thus, valid text file pages always end with a carriage return, possibly followed by nulls.

NOTE: A number of communications packages in common use under the p-System violate the restriction against lines crossing page boundaries.

NOTE: Much of the software that works with text files takes an occurrence of a null byte as a signal that there is no more text present on that page. Therefore care should be taken not to create a text file containing a null embedded within the text. The remainder of the text following the null may be ignored.

Many strings of consecutive blank characters occur in text files. In text containing typically indented Pascal programs these strings of blanks tend to occur most frequently at beginnings of lines. To conserve space in textfiles the p-System editors employ a blank compression sequence. This sequence reduces strings of blanks occurring at the beginning of a line to two bytes.

When the ASCII DLE character (decimal 16) occurs as the first character of a line the byte following it indicates the number of leading blanks contained on that line. This value is 32 more than the actual number of leading blanks; the offset of 32 brings the leading blank count into the range of displayable ASCII characters. Valid text files have DLE compression sequences *only* at the beginning of a line, never in the middle of a line.

NOTE: In current p-System releases certain system utilities generate text files with DLE sequences embedded in the middle of lines (the Assembler is an example of such a utility).

Blanks occurring at the end of a line (prior to the return character which terminates the line) are valid characters and are not compressed. However, certain editor functions strip these trailing blanks.

NOTE: The p-System editors can often be used on textfiles that violate the restrictions against lines crossing page boundaries and DLE sequences appearing in the middle of lines. Often, the updated file is forced by the editor to a proper textfile format.

When the tab key is used in an edit session a string of blanks is inserted into the file. The screen-oriented editors never insert the tab character itself (a byte containing decimal 9). Tab characters within text files are not supported by the p-System editors and make a text file almost impossible to edit.

8.0.2 Header

Headers are two-block records that precede the text of valid text files. The information in the header is used by the system editors and generally stripped and ignored by the rest of the system.

Most of the 1024 bytes in the header remain unused by the current editors and are filled with nulls. The first word of the header is usually filled with a non-null value by the editor when a text file is initially edited. Some versions of the editor use this field to indicate editor version. The remainder of the header contains environment information, such as marker names and positions, tabstops and auto-indent and filling modes. Default values for these fields are assigned by the editor and may be changed by the user with the editor's S(et E(nvironment command.

NOTE: The header record format is not uniform between the various editors currently in use under the p-System. Thus the environment information of a file created under one version of the editor may be lost when the file is edited under a different version of the editor.

NOTE: Header records of many editor releases are not portable to machines of the opposite byte sex. Thus, when editing a text file on a machine of opposite byte sex from the one upon which the file was created, environment information can be lost.

At times, files containing text may be imported to the p-System environment without header records. These files cannot be handled as-is by the editors since the first two blocks of their text will be taken as a header record. One can often edit such a file by appending two blocks containing nulls to its beginning. Difficulties may remain if the body of the file violates the restrictions mentioned in the previous section.

It is tempting for authors of word-processing software to use the header fields unused by the editor for their own data. For example, a text formatter may embed formatting instructions in the header of a text file the first time it is processed so that the information need not be re-entered by the user when the file is again printed at a later date.

Unfortunately, this trick will not work with all versions of the editor. While some versions leave the "unused" header fields intact, other versions reset the unused fields to nulls when the edited file is written. Thus, the only safe procedure in such a situation is to use a separate file for the application's data.

8.1 Using SCREENOPS Data Structures and Procedures

Portability is the p-System's reason for being. The issue of portability must be addressed not only on the level of a machine's instruction set and I/O subsystem but also on the level of handling the terminal's editing and cursor control functions.

The system itself, and applications that run under it, must often perform such functions as clearing a line or the entire screen, moving the cursor to a specific set of screen coordinates, or repositioning it relative to its current location.

Most terminals employ control or escape sequences of one or more characters which are taken not as displayable characters but as editing or cursor movement instructions. Unfortunately, these sequences differ from terminal to terminal. Therefore applications including these sequences explicitly are limited to functioning with the terminal for which they were written.

When the p-System is initially configured for a particular system a utility called SETUP is run which enables the user to specify the sequences peculiar to the terminal in use. SETUP creates a file which contains these terminal parameters (among other things). This file, when given the name SYSTEM.MISCINFO, is read into memory each time the system is booted.

When the system needs to perform a terminal-specific function (such as clearing the screen and homing the cursor) it refers to the MISCINFO information to determine precisely what to send to the screen to cause the desired action to take place. Thus, the code itself is independent of the terminal being used.

It is possible (and sometimes necessary) for application programs to refer to the memory locations containing the MISCINFO information. As was discussed at the beginning of this chapter, however, such low-level access to system data structures is risky. There is the danger that the data itself become modified, thereby corrupting the system. There is the further danger that the data structure change in a later system release, rendering the program performing low-level access obsolete.

The SCREENOPS unit supplied with Version IV of the p-System (subset implementations for other p-System versions are available in the USUS Library) provides a high-level interface to terminal control functions. The programmer may refer to a set of standard procedures and functions to accomplish basic cursor movement and screen editing.

SCREENOPS also contains additional amenities in the form of procedures which ease some of the mundane chores common in programs performing screen I/O. These include screening keyboard input for a specified set of valid characters, generating prompt lines similar to the ones used by the operating system itself and other useful functions. SCREENOPS provides the ability to discover miscellaneous MISCINFO information such as general terminal characteristics.

NOTE: The p-System documentation refers to a "text port", which is a rectangular subsection of the screen. The SCREENOPS procedures are defined to operate on text ports. However, current versions of the p-System do not support other than full screen operations.

It is recommended that the reader have available a listing of the interface section of the SCREENOPS unit while reading the remainder of this section. This can be obtained using the DECODE utility, or by compiling the example which follows.

The behavior of many of the routines in SCREENOPS is evident from their nomenclature. The `Sc_Clr_Cur_Line`, `Sc_Clr_Line`, `Sc_Clr_Screen`, `Sc_Erase_to_EOL` and `Sc_Eras_EOS` procedures all cause the reaction on the screen described by their names. (Note the common convention of beginning the names of all interface section entities with the same character sequence; "`Sc`" in this case. This convention makes "imported" entities easier to identify in a host program).

Similarly, the `Sc_Left`, `Sc_Right`, `Sc_Up`, `Sc_Down` and `Sc_Home` procedures all cause the cursor to move in the prescribed manner.

When the p-System is initially configured a GOTOXY procedure is written and linked to the operating system. This permits the user to customize direct cursor addressing for the terminal in use. SCREENOPS contains a procedure called `Sc_Goto_XY` which the programmer may use to move the cursor to a desired set of screen coordinates in a portable manner. `Sc_Goto_XY` uses the customized GOTOXY procedure bound into the operating system.

The system is capable of tracking the current cursor location if the programmer does not UNITWRITE to the screen and if direct cursor addressing is *always* accomplished using `Sc_Goto_XY` (if the programmer explicitly writes a terminal-dependent direct cursor addressing sequence to the screen the system will have an erroneous view of the current cursor location.)

The current cursor coordinates can be determined using the `Sc_Find_X` and `Sc_Find_Y` functions.

`Sc_GetC_Ch` is a procedure which continually reads and rejects characters from the (non-echoing) keyboard until a desired character is entered. The second parameter to this procedure is the set of desired characters. The first parameter is a VAR parameter of type char which `Sc_GetC_Ch` fills with the accepted character. Lower case characters are automatically converted to upper case before `Sc_GetC_Ch` attempts to match them to the set.

This procedure would typically be used after a prompt line or a menu is presented to the user, where the acceptable response is a single character to select the appropriate menu item. `Sc_GetC_Ch` can be used to easily "filter" out illegal responses. The returned character would then be used as the selector in a case statement containing a case label for each of the elements in the desired set. (See the use of `Sc_Prompt` in the example below. `Sc_GetC_Ch` provides a subset of the functionality of `Sc_Prompt`.)

`Sc_Space_Wait` (called `Space_Wait`, without the "Sc", in some implementations) writes the legend "Type <space> to continue" to the terminal. It then reads and rejects characters from the (non-echoing) keyboard until either a <space> or an <escape> character is entered. This function returns TRUE if an <escape> was typed and FALSE if a <space> was typed.

Typically, `Sc_Space_Wait` appears as the conditional of an IF statement when an error or other exceptional condition is noted. The user would be presented with one or more lines of error message, and `Sc_Space_Wait` would allow the user to read the message before the screen was erased. Since <escape> is frequently used to abort a program entirely, the IF statement might read

```
If Sc_Space_Wait(TRUE)
  then exit(program)
```

```
else
{provide for a retry or chain to another program, etc}.
```

`Sc_Space_Wait` has a single boolean parameter. If TRUE, a UNITCLEAR is issued to the keyboard before the legend appears. This has the effect of clearing the keyboard buffer in case the user had typed ahead without anticipating an error message. If the parameter is FALSE the keyboard buffer is not cleared. Its contents will be taken as input by `Sc_Space_Wait`.

`Sc_Prompt` is an elaborate function that enables the programmer to easily use a prompt line in the precise style of the system prompts. The first parameter to `Sc_Prompt` is a string containing the prompt line, which may be up to 255 characters long. As much of the prompt as will fit on a single line is displayed; `Sc_Prompt` allows the user to cycle through the rest of the prompt, in line-sized segments, by typing "?". When the entire prompt has been displayed "?" returns the user to the original line.

If a colon (:) appears in the string, characters to its left will appear at the beginning of *every* segment of the prompt (this is standard operating system usage). If a set of brackets appears at the end of the string it will appear at the end of every prompt segment. (The brackets would typically be used to contain a program version number.)

The programmer may specify the character used to define the points where the string is broken into the prompt segments; the comma (,) is normally used. As `Sc_Prompt` cycles through the segments it automatically generates a "?" character at the end of all but the last prompt segment.

`Sc_Prompt` not only displays the prompt line, it can also read the keyboard, rejecting all but the correct responses. In this respect it functions identically to `Sc_GetCh`, described above. A boolean is used to indicate whether or not `Sc_Prompt` will be used for this additional function.

The programmer has the option of positioning the prompt at any point on the screen. Once a prompt segment has been written the programmer has the option of positioning the cursor at any screen location or allowing `Sc_Prompt` to leave the cursor immediately following the prompt segment.

See the example below for the precise sequence of `Sc_Prompt` parameters.

`Sc_Check_Char` is a function which examines the last character read from the terminal and returns TRUE if that character was a <backspace> or <del> character. This function would normally be used in a loop which reads in characters to a string one at a time.

This function has three VAR parameters. The first is of type `Sc_Window`, which is a PACKED ARRAY[0..0] OF CHAR. The variable passed as this parameter might be declared as a variant record, where one variant is the PACKED ARRAY[0..0] and the other variant is the string whose value is being read. The packed array is a device whereby `Sc_Check_Char` can access the entire string as an array without knowing its size (without range checking).

The second parameter is an index to the position in the string containing the character just read. The third parameter is an integer indicating the number of characters remaining to be read in the string.

The programmer would initialize the string to blanks, the index to one, and the count of characters remaining to the size of the string. In a REPEAT loop, the indexed character of the string would be read. `Sc_Check_Char` would then be invoked as the condition of an IF statement. If the character read were not a <backspace> or <del> `Sc_Check_Char` would increment the index, decrement the count of remaining characters and return FALSE. The UNTIL clause would end the loop when this count reached zero.

But if the character read *were* a <backspace> or <del> the index would be set back (by one for a <backspace> and to the beginning of the string for a <del>), the count of remaining characters would be increased (by one for a <backspace> and to the size of the string for a <del>), and `Sc_Check_Char` would return TRUE. The THEN clause of the IF statement would be executed; this would probably be a statement to reset the position of the removed character to a blank.

Note that `Sc_Check_Char` itself neither reads into nor modifies the contents of the string. However, the screen is properly handled and need not be manipulated by the programmer; <backspace> removes the last character typed from the screen and <del> removes the entire entry.

`Sc_Map_CRT_Command` is perhaps the handiest function in SCREENOPS. This function scans the character passed to it to identify which key command it is. The possible key commands include backspace, escape, ETX and the arrow keys, among others (see type `Sc_Key_Command` in the SCREENOPS interface section). Note that any of these might be prefixed. Without this function the programmer wishing to write a portable program would have to examine the MISCINFO information to determine whether or not each key was prefixed, and have code in the program to check for the appropriate prefix (as well as code to handle no prefix).

`Sc_Map_CRT_Command` does all this automatically. If the character passed to `Sc_Map_CRT_Command` matches a key command as it stands, with no prefix, the function returns that key command (an element of `Sc_Key_Command`). If not, the function assumes that the character passed to it is a prefix. It reads another character from the keyboard, and attempts to match that prefix-character combination to a key command. If a match is found, that key command is returned. If not, the value `Sc_Not_Legal` is returned.

Typically, `Sc_Map_CRT_Command` would be preceded immediately by a

```
READ(KEYBOARD,CH);
```

and followed by a case statement with case labels for each `Sc_Key_Command` element expected by the program at that point.

Function `Sc_Screen_Has` is a boolean function which enables the programmer to discover whether or not the terminal in use has a particular screen function. Function `Sc_Has_Key` is a similar function which enable the programmer to determine if the keyboard can produce a specified key command.

`Sc_Use_Info` utilizes a data structure of `Sc_Info_Type` to pass information back and forth between a program and SCREENOPS. This structure contains fields corresponding to many pieces of MISCINFO information including the screen height and width, scrolling abilities and speed characteristics. The first parameter to `Sc_Use_Info` specifies the direction of the information transfer. The second parameter to the procedure is the structure itself. Changes to screen characteristics accomplished through `Sc_Use_Info` are local to the program making them.

Example of SCREENOPS usage:

```
program scdemo;
{$L PRINTER:}
uses {$U SCREENOPS.CODE} screenops;
var
  s1, s2: string;
  big: sc_long_string;
  ch: char;
  name: packed record
          case integer of
            1:(pname: string[15]);
              {String for Sc_Prompt input }
            2:(nname: packed array[0..0] of char);
              {Dummy overlay for Sc_Prompt}
          end;
  MyScreen: sc_info_type;
  high, wide, index, remaining: 0..255;
  x, y: integer;

procedure DoArrows;
const
  HellFreezesOver = false;
var
  GetOut: boolean;
  ch: char;
  direction: sc_key_command;
begin
  sc_goto_xy(0,0);
  sc_clr_cur_line;
  write('Arrows: Arrow keys move, <esc> escapes');

  GetOut := false;
  sc_goto_xy(x,y);
  repeat
    read(keyboard, ch);
    direction := sc_map_crt_command(ch);
    case direction of
      sc_up_key   : if y > 0 then sc_up;
      sc_down_key : if y < high then sc_down;
      sc_left_key : if x > 0 then sc_left;
      sc_right_key: if x < wide then sc_right;
      sc_escape_key: GetOut := true;
    end;
    x := sc_find_x;
   {preserve new x and y values for possible
    further manipulation}
    y := sc_find_y;
    if GetOut
      then exit(DoArrows);
  until HellFreezesOver;
end;

begin
  sc_use_info(sc_get, MyScreen);
 {determine screen dimensions}
  high := MyScreen.misc_info.height;
  wide := MyScreen.misc_info.width;

  sc_clr_screen;
  write('Hi!  What is your name? ');
  index := 1;
  remaining := 15;
  name.pname := '               ';
 {if uninitialized will remain at length 0!}
  repeat
    read(name.pname[index]);
    if sc_check_char(name.nname, index, remaining)
      then fillchar(name.pname[index], remaining, ' ');
     {blank out deleted character(s) }
    if eoln
      then index := index - 2;
     {remove spurious EOLN characters from string}
  until (remaining = 0) or eoln;
  name.nname[0] := chr(index);
 {diddle the length byte to reflect true length}

  writeln;
  writeln('Welcome to some nonsense, ',name.pname,'!');
  writeln('Let me know when you''re ready to begin.');
  if sc_space_wait(true)
 {user hit escape key (can you blame him?)}
    then exit(program);
  s1 :=
   'Fun: A(rrow keys, M(iddle of screen, Top L(eft, Top ';
  s2 := 'R(ight, Bottom leF(t, Bottom RigH(t, Q(uit [0.0a]';
  big := concat(s1,s2);
  x := -1;
 {force cursor to end of prompt, first
```

```
     time Sc_Prompt is invoked}
     y :=  0;
     repeat
       ch := sc_prompt(big,  {prompt line}
                        x,y,
       {coords of cursor following display of prompt segment}
                        0,0,
       {coords of each prompt segment beginning}
                        ['A','M','L','R','F','H','Q'],
       {acceptable input chars}
                        false,
       {we do want a returned character}
                        ',');
       {break prompt into segments at a ','}
       case ch of
         'A': DoArrows;
         'M': begin
                x := wide div 2;
                y := high div 2;
                sc_goto_xy(x,y);
              end;
         'L': sc_goto_xy(0,0);
         'R': sc_goto_xy(wide,0);
         'F': sc_goto_xy(0,high);
         'H': sc_goto_xy(wide,high);
       end;
       x := sc_find_x;
      {preserve new x and y values for possible
       further manipulation}
       y := sc_find_y;
    until ch = 'Q';
  end.
```

8.2 COMMANDIO Monitor and I/O Redirection

The system unit COMMANDIO contains a number of documented subroutines, which we have already seen. These are CHAIN (section 4.4), EXCEPTION (section 4.9) and REDIRECT (section 4.29).

COMMANDIO also contains a number of additional variables and procedures which relate to I/O redirection and use of the monitor. Note that COMMANDIO routines are not available in pre-Version IV p-Systems.

The following boolean variables are available to a program which USEs the COMMANDIO unit: `HaveChain`, `InRedirect`, `OutRedirect`, `MonitorOpen` and `InMonitor`.

`HaveChain` is TRUE when a chain is pending – that is, if control will be passed to another program after the current program completes execution rather than the system promptline being displayed. `HaveChain` is FALSE otherwise. If a program must take different courses of action depending on whether it will return control to the user or to another program it can use `HaveChain` as the basis for the decision. Typically, this situation can arise when the conditional CHAIN appeared earlier in the program. `HaveChain` will determine whether the CHAIN was actually executed or not. An example using `HaveChain` appears at the end of this section.

`InRedirect` is TRUE when input has been redirected away from the standard input device (normally the console). It is FALSE otherwise. This variable would be examined by a program if it needed to know the source of its input. For example, a program might employ screen control if user input were to come from the console CRT. The program would avoid setting up a screen, however, if user input were redirected to a teletypewriter.

`OutRedirect` is TRUE when output has been redirected away from the standard output device (normally the console). It is FALSE otherwise. A different format might be used for a report redirected to the printer, instead of to the console. A program can determine whether the output had been redirected by examining `OutRedirect`.

`MonitorOpen` is TRUE if a monitor file is currently open, FALSE otherwise. Recall that Version IV of the p-System permits a "recording" to be made of a work session; those keystrokes recorded can be "replayed" without being rekeyed at a later date. The keystrokes are recorded in a monitor file, which may be opened using the M(onitor command at the main system promptline or the `startmonitor` procedure explained in this section. `MonitorOpen` can be examined by a program to determine whether or not such a recording file has in fact been opened.

`InMonitor` is TRUE if the monitor file is currently active. The recording process may be temporarily suspended – though the monitor file be left open – using the M(onitor command at the main system promptline. Thus, selected keystrokes can be left out of the monitor file. Recording can be resumed into the file at any time with the M(onitor command. When monitoring is suspended `InMonitor` is FALSE.

The procedure `StartMonitor` performs the equivalent of the M(onitor command of the main system promptline. The user is prompted for a name for the monitor file, which is then opened.

NOTE: `InMonitor` remains FALSE even after `StartMonitor` is executed. If it is desired not only to open a monitor file but also permit recording within it, the program must explicitly set `InMonitor := true` after the call to `StartMonitor`.

The procedure `StopMonitor` closes the monitor file. No further recording will be done to the monitor file.

NOTE: `StopMonitor` does not affect the value of `InMonitor`. Even if a previously open monitor file is closed with `StopMonitor`, `InMonitor` will remain TRUE. The user *must* explicitly set `InMonitor := false` after the call to `StopMonitor` else the system is in danger of crashing.

`StopMonitor` has a single boolean parameter. If its value is TRUE the monitor file will be closed and retained. If its value is FALSE the monitor file will be closed and discarded.

The following program uses `HaveChain` to enable a rudimentary program call facility. "Program call" means the capability for a program to call another program at some point, for control to be immediately tranferred to the called program, and for execution of the calling program to resume at the point *after* the call instruction once the called program completes.

The entire program `CallDemo` is divided between the THEN and ELSE halves of an IF statement. The part of the program that precedes the call, and the call itself, are in the THEN half. The part of the program that is to be executed after the call completes appears in the ELSE half. The condition of the IF statement is the truth of `HaveChain`. If `HaveChain` is FALSE then we have not yet executed any chain instructions. This indicates that we are beginning the program, not returning to it after a call. Therefore if `HaveChain` is FALSE we execute the THEN half of the IF statement, including the call to the called program in the form of a chain, and *two* chains back to `CallDemo`!

Since the IF statement is satisfied the program completes. Recall that consecutive chains are stacked and executed in a first-in, first-out fashion. So control will be passed to `CalledDemo`– the called program. Note that `CalledDemo` need not be aware of its role as a called program. It is an ordinary program.

When `CalledDemo` concludes, the next (stacked) chain to `CallDemo` is executed. But there is still one more chain to `CallDemo` pending since it had been chained-to twice. Therefore `HaveChain` is TRUE and the ELSE half of the IF statement is executed. `Exception` is called upon to cancel the remaining call to `CallDemo`, which was present only to keep `HaveChain` TRUE. The remainder of the program is executed.

```
program CallDemo;
uses {$U COMMANDIO.CODE} commandio;
begin
  if not havechain
    then begin
      {This is the section of code executed
       before "call" of CalledDemo}
      writeln('We are in calling program, before "call".');
      chain('CalledDemo'); {The actual "call"}
      chain('CallDemo');   {Gets us back to this program}
      chain('CallDemo');   {Assures that we end up
```

```
                                     in "after" section}
    end                    {by forcing havechain to true}
    else begin
      exception(true);
     {turns off chaining to avoid infinite regress}
     {This is the section of code executed after
      "call" of CalledDemo}
      writeln('We are in calling program, after "call".');
    end;
end.


program CalledDemo;
begin
  writeln('   We are in "called" program!');
end.


Output:

We are in calling program, before "call".
   We are in "called" program!
We are in calling program, after "call".
```

8.3 The KERNEL

Much of the global information maintained by the operating system is accessible to user programs as well. Section 5.12 described how this information is made available to a program under various versions of the p-System. In this section a version IV or later p-System is assumed; thus, the global data structures, constants, variables and routines are made available by USEing KERNEL.CODE from within the host user program.

The global information changes from one version of the p-System to another, often to a considerable degree. Nevertheless, a good part of what is said here is applicable to earlier versions of the p-System as well as to Version IV. A pre-Version IV program can gain access to the system globals either with the {$U-} compiler directive (section 5.12) or by explicitly declaring the required items. This is particularly true with regard to accessing volume directories, as described in section 8.3.3.

The system global variables normally reside in low memory. When KERNEL is used, however (or {$U-} in pre-Version IV p-Systems), the programmer may access these items by name.

Therefore, programmers intending to use the system globals from within user programs are strongly urged to acquire a listing of the globals for their version of the system before proceeding. Listings of the globals for some p-System versions can be obtained from the USUS Library. A listing of the globals for the p-System version in use by the programmer can be had by using the DECODE utility to obtain the interface section of KERNEL.CODE or by requesting a compiler listing of a program USEing KERNEL.CODE.

Section 8.3.1 discusses some of the constants available in the system globals. Section 8.3.2 describes how the global variable containing the system date may be accessed. In Section 8.3.3 a method for accessing disk directories from within user programs is presented. This method is independent of p-System version.

8.3.1 System Constants

The system constants fall into three categories: those that describe fixed limits of the current version of the OS (such as maximum number of files permissible in a disk directory or number of characters in a file name), those that define values for error codes (such as unknown system error or segment fault) and those that define commonly used values (such as the ASCII codes for EOL or DLE).

The system constants are documented briefly in comments included in the listing of the globals. We will discuss them as they arise in the following sections.

8.3.2 Accessing the System Date

The first example of accessing a system global will be to check the system date. As discussed in section 8.3.2, the latest date set is stored in the directory of the system disk. At the next boot that value is read in and stored in a global variable called `THEDATE` which is of type `DATEREC`. The system date may be set using the Filer D(ate command. This updates both the global variable `THEDATE` and the date on the system volume.

The global type `DATEREC` appears as follows:

```
DATEREC = PACKED RECORD
   MONTH: 0..12;
             (*0 IMPLIES DATE NOT MEANINGFUL*)
   DAY: 0..31;
             (*DAY OF MONTH*)
   YEAR: 0..100;
             (*100 IS TEMP DISK FLAG*)
               END (*DATEREC*) ;
```

This program illustrates how the globals are accessed under p-System Version IV.1. The file KERNEL.CODE (here assumed to reside on the default volume) is USEd and the identifiers appearing therein may immediately be accessed. `THEDATE` is checked for validity and edited for display.

The program could easily be modified so that it not only displayed the date but also permitted its modification since the global variables may be changed at will. It might then be used as a SYSTEM.STARTUP to enforce daily date changes. For a guide to modifying the date as stored on the system volume see section 8.3.3.

```
program DatePlay;
uses {$U KERNEL.CODE} kernel;

const
  century = '19';
var
  ch: char;
begin
  write('Current date is ');
  with THEDATE do begin
    if (month = 0) or (day = 0) or (year = 100)
      then write('not meaningfull')
      else begin
        case month of
          1: write('January ');
          2: write('February ');
          3: write('March ');
          4: write('April ');
          5: write('May ');
          6: write('June ');
          7: write('July ');
          8: write('August ');
          9: write('September ');
         10: write('October ');
         11: write('November ');
         12: write('December ');
        end;
        write(day, ',');
        writeln(' ', century, year:2);
      end  {else};
  end  {with};
end.
```

8.3.3 Using Directories from Programs

The UCSD file system provides sufficient support to render it unnecessary for an application to explicitly manipulate disk volumes and their directories under normal circumstances. But there are occasions when it is helpful to be able to perform such manipulations. It may be helpful to condense a volume from within an application, for example (by moving the files together so that all free space is at the end of the volume).

Later versions of the p-System include standard units which facilitate this sort of thing. These should certainly be used whenever possible. There will probably be times, though, that the programmer may wish to perform these manipulations explicitly. This will certainly be true of those who use pre-Version IV implementations of the p-System.

Before continuing with this section see chapter 7 for a discussion of the UCSD file system; in particular see section 7.2.6 for a discussion of disk directories.

8.3.3.1 Reading and Modifying Directories

In current p-System implementations a p-System directory is limited to containing 77 entries. This value is defined by the global constant `MAXDIR`. Each directory entry contains information about a file residing on that volume and is described by the global structure `DIRENTRY`. The entire directory is an array of 78 `DIRENTRY`s. The first element of this array, which is customarily indexed as the 0'th element, contains information about the disk volume itself. Thus, `DIRENTRY` is a variant record with one variant for the volume information and another for file information.

The structure of the directory has changed little from one version of the p-System to another. Directories are normally compatible across all p-System versions.

A disk directory is stored beginning at (zero-based) block 2 of the volume. This value is recorded in the global constant `DIRBLK`. Unit I/O may be utilized to read the volume directory by absolute block number, but this requires that the volume number be known to the program. For many applications this is not the case; the volume name is known but the volume may be mounted on any drive. Therefore BLOCKREAD/BLOCKWRITE are used to access the disk directory.

These intrinsics require an untyped file as their first operand. However, an untyped file may be RESET with a volume name (rather than the usual file name) as in

```
reset (diskFil, VOL);
```

where `diskFil` is declared as type FILE and `VOL` is a string containing the volume name. Thus, any part of the disk may be accessed by absolute block number using the block I/O intrinsics on `diskFil`, including the area containing the directory.

The directory will be BLOCKREAD into an array of `DIRENTRY`s. But BLOCKREAD will only read an integral number of blocks. The usual array of 78 `DIRENTRY`s contains 2028 bytes (`SIZEOF(DIRENTRY)` is equal to 26; 26 x 78 = 2028). Reading three blocks for the directory takes in 3 x 512

= 1536 bytes, which is too few; part of the directory will be ignored. Reading four blocks yields 4 x 512 = 2048 bytes, which will take in the entire directory but will also read in the 20 bytes beyond the directory. The problem is that if the array of `DIRENTRYs` contains the usual 78 elements, memory beyond the array will be trashed since BLOCKREAD does no bounds-checking.

The solution is to declare an array of 79 `DIRENTRYs`, with the last entry there for no other reason than to allow enough room for a BLOCKREAD of 4 complete blocks (79 x 26 = 2054; more than enough for the 2048 bytes in 4 blocks). The first of the 79 `DIRENTRYs` will contain volume information. The following 77 may contain information about the files on the volume. The last of the 79 entries will contain garbage; it is the programmer's responsibility to assure that this element is never accessed.

The program which follows demonstrates how one can read and/or write a directory from within a program. The heart of the program is the routine `UseDirInfo` which reads the directory and either returns or replaces (depending on the function specified in the `fun` parameter) values for the file specified. In this illustration the values manipulated are file size and last byte; these can easily be changed to any or all of the file characteristics stored in the directory.

```
program DirDemo;
const
     MAXDIR = 77;
   (*MAX NUMBER OF ENTRIES IN A DIRECTORY*)
     VIDLENG = 7;
   (*NUMBER OF CHARS IN A VOLUME ID + 1*)
     vidlengPlus1 = 8;
   (*room for colon between VID and TID*)
     TIDLENG = 15;
   (*NUMBER OF CHARS IN TITLE ID*)
     FBLKSIZE = 512;
   (*STANDARD DISK BLOCK LENGTH*)
     DIRBLK = 2;
   (*DISK ADDR OF DIRECTORY*)

type
     DATEREC = PACKED RECORD
                 MONTH: 0..12;
               (*0 IMPLIES DATE NOT MEANINGFUL*)
                 DAY: 0..31;
               (*DAY OF MONTH*)
                 YEAR: 0..100;
               (*100 IS TEMP DISK FLAG*)
        END (*DATEREC*) ;

     VID = STRING[VIDLENG];

     DIRRANGE = 0..MAXDIR;
     TID = STRING[TIDLENG];

     FILEKIND = (UNTYPEDFILE,XDSKFILE,CODEFILE,TEXTFILE,
   INFOFILE,DATAFILE,GRAFFILE,FOTOFILE);

     DIRENTRY = RECORD
                  DFIRSTBLK: INTEGER;
                (*FIRST PHYSICAL DISK ADDR*)
                  DLASTBLK: INTEGER;
                (*POINTS AT BLOCK FOLLOWING*)
                  CASE DFKIND: FILEKIND OF
                    UNTYPEDFILE:
                  (*ONLY IN DIR[0]...VOLUME INFO*)
                       (DVID: VID;
                      (*NAME OF DISK VOLUME*)
                        DEOVBLK: INTEGER;
                      (*LASTBLK OF VOLUME*)
                        DNUMFILES: DIRRANGE;
                      (*NUM FILES IN DIR*)
                        DLOADTIME: INTEGER);
                      (*TIME OF LAST ACCESS*)
      XDSKFILE,CODEFILE,TEXTFILE,INFOFILE,
      DATAFILE,GRAFFILE,FOTOFILE:
                       (DTID: TID;
                      (*TITLE OF FILE*)
                        DLASTBYTE: 1..FBLKSIZE;
                      (*NUM BYTES IN LAST BLOCK*)
                        DACCESS: DATEREC);
                      (*LAST MODIFICATION DATE*)
END   (*DIRENTRY*) ;

      BlkRange = 1..FBLKSIZE;
      vidstring = string[vidlengPlus1];
      DirFunct = (GetInfo, PutInfo);
      dir = array [0..78] of DIRENTRY;
var
      SrcLB, SrcSiz, bomb: integer;
      MyVol: VIDSTRING;
      MyFil: TID;

procedure MakeUpper(var s: string);
var
  i: integer;
begin
  for i := 1 to length(s) do
    if s[i] in ['a'..'z']
      then s[i] := chr(ord(s[i]) - ord('a') + ord('A'));
end;

procedure UseDirInfo (fun : DirFunct;
                         VOL : VIDSTRING;
                         FIL : TID;
                     var LB  : BlkRange;
                     var SIZ : integer;
                     var code: integer);
    var
       directory : dir;
       diskFil   : File;
       int       : integer;
       index     : 0..78;
       found     : boolean;

    procedure scram (CodeVal : integer);
    begin
       code := CodeVal;
       exit (UseDirInfo);
    end;

    begin
       code := 0;
       reset (diskFil, VOL);
       int := blockread (diskFil, directory, 4, DIRBLK);
       close (diskFil);
       if ((ioresult <> 0) or (int <> 4))
          then scram(1);
       if directory[0].dnumfiles > 0
          then begin
             index := 1;
             found := false;
             repeat
                if directory[index].dtid = FIL
                   then found := true;
                index := succ (index);
             until (found or (index = 78));
             if not found
                then scram(2);
          end;
       index := pred (index);
       with directory[index] do begin
          if fun = GetInfo
             then begin
                LB := dlastbyte;
                SIZ := dlastblk - dfirstblk;
             end
             else begin
                dlastbyte := LB;
                reset (DiskFil, VOL);
                int :=
                  blockwrite(DiskFil, directory, 4, DIRBLK);
                close (diskFil);
                if ioresult <> 0
                   then scram(3);
             end;
       end;
    end  {UseDirInfo};

begin
  writeln(
'Directory Access Demo - Finds last byte and size of a file'
                                                          );
    writeln;
    write('Name of file: ');  readln(MyFil);
    write('Volume: ');  readln(MyVol);
    MakeUpper(MyFil);
    MakeUpper(MyVol);
    UseDirInfo(GetInfo, MyVol, MyFil, SrcLB, SrcSiz, bomb);
    case bomb of
      0: writeln('The last byte is ', SrcLB,
                 ' and the size is ', SrcSiz);
      1: writeln('Error reading directory!');
      2: writeln('File not found!');
    end;
    writeln(sizeof(DIRENTRY));
    writeln(sizeof(DIR));
  end.
```

Note that the program above explicitly declares those KERNEL structures it needs. Thus it does not have to use KERNEL.CODE and is version-independent.

DIRENTRY begins with two integers, DFIRSTBLK and DLASTBLK, which point to the first block of the directory item and the block immediately following the directory item, respectively. The total size of the file is equal to DLASTBLK − DFIRSTBLK. The record continues with two variants.

The first variant applies to the volume itself and appears in only one DIRENTRY, the first (usually addressed as the 0'th element of the array of DIRENTRYs). It contains the DVID, the volume ID (normally not needed by the program since the volume ID was used to obtain the directory in the first place). It also contains the last block number of the volume and DNUMFILES, the number of files on the volume. The latter value is important since, if information about a particular file is desired, the program will have to search for it through the array of 77 DIRENTRYs. DNUMFILES must be used as the index of the last array element to be checked. All DIRENTRYs past the one indexed by DNUMFILES will contain garbage. The last two elements of the first variant are two integers representing the time of last access and DLASTBOOT which, on a system disk, is the latest date set. It is DLASTBOOT (as well as the global variable THEDATE) which is changed by the filer D(ate command.

The second variant of DIRENTRY applies to the directory entries of each of the files on the volume. There are DNUMFILES of these, where DNUMFILES is less than or equal to 77. Each entry contains the following variables:

DTID, a string with the file name (including the "." and extension if present). This entry is normally used as the key when searching for a particular file.

DLASTBYTE, an integer representing the number of bytes in the last block of the file. This value is important when it is desired to BLOCKREAD a file into a buffer, then remove individual records from the buffer. A file always contains an integral number of blocks, but the last block will usually contain garbage beyond the last record. DLASTBYTE indicates the last byte in that block which contains meaningful information. Thus, when removing records from the buffer, DLASTBYTE will be used to determine the end boundary of the last record in the file. (DLASTBYTE will always have the value 512 for textand code files since the editor and compiler fill the entire last block of the file.)

DACCESS, which is of type DATEREC, and indicates the last modification date of the file. The system updates DACCESS automatically based on the value of THEDATE.

8.3.3.2 Duplicate Directories

As discussed in section 7.2.6.1, it is possible for a p-System disk volume to have two copies of its directory. In this manner it is possible to restore a disk volume with a corrupt directory using a utility which recreates the directory from the duplicate copy. The duplicate directory, when present, immediately follows the standard directory.

When a disk volume is initialized the user is queried as to whether a duplicate directory is desired. If a duplicate directory is present the system automatically updates the duplicate when it updates the standard directory.

Programs which explicitly modify disk directories should update the duplicate directory at the same time, if present, to preserve the duplicate's value as a backup.

The duplicate directory may be read and modified in the same fashion as the standard directory. It has an identical structure. But how does one determine if a disk volume has a duplicate directory?

Whether a disk volume has a duplicate directory can be ascertained by examining the DLASTBLK entry for the 0'th element in the array of DIRENTRYs. On disk volumes with no duplicate directory the value will be 6. When a disk volume has a duplicate directory the value will be 10. In the latter case the disk volume has a duplicate directory. It should be updated along with the standard directory.

8.4 Segment Code Management

This section discusses how the operating system manages and executes code files. The structure of code files, as well as the algorithms used by the p-System to read code files from disk into memory for execution, are covered in some detail.

The internal structure of code files and code file management are normally transparent to both the user and the programmer. Most of the time there is no reason for either to be aware of what transpires when a program is executed. It is discussed here for two reasons: First, there probably will be rare occasions when information on this subject *is* helpful in the development of an application. Second, it is assumed that the reader, like the authors, has a full measure of curiosity regarding such a frequently used tool as an operating system.

As the p-System has evolved it has become progressively more sophisticated. Although what happens when a program is executed appears superficially the same from the earliest releases of the p-System (Version I.3) to the current SofTech release (Version IV.12), the structure of the code file and the methods used to manage it are very different. It is likely that future, more capable releases will be different still.

The various p-System suppliers used different releases of the p-System as their starting point and extended their products in different directions. Thus, SofTech's IV.12 is different from Apple Computer's latest release, which is different from the latest Western Digital release, and so on.

The focus will be on how Version IV of the p-System maintains and executes code files. Be forewarned that for different versions of the p-System details of this discussion will probably not be applicable, and newer releases of Version IV p-System may render details of this discussion obsolete.

To avoid obscuring explanations with too much material, less common situations such as how native code or assembly language procedures relate to the code management scheme are not discussed.

It is recommended that this section be read (regardless of p-System version) for an appreciation of the "large picture". Once this is obtained, the reader may refer to the Internal Architecture Guide (or equivalent manual) supplied as part of the p-System documentation.

8.4.1 Introduction and Overview

Here is a simplistic picture of the process of preparing and executing a program: The programmer writes the program as a single text file. The program is compiled into a single code file. The code file is loaded into memory and executed.

There are two basic reasons why this simplistic approach is unrealistic. First, a program compiled to a single chunk of code will often be too large to fit into the memory of a typical microcomputer. Second, requiring that a program exist as a single large whole is wasteful. Entirely different programs often have a large body of common code. Many functions (such as screen handling, file management, etc.) are present in virtually all programs. If programs were maintained as single whole entities much of the auxiliary storage would be taken up by duplicated code.

The alternative approach to code management taken by the p-System is to permit units of code smaller than an entire program. These units are called segments. Under the p-System, a segment is a code "atom"; its components can be examined but not physically separated.

Program code is composed of one or more segments. It is not necessary for all the segments of a program to be resident in memory at the same time. The operating system automatically loads segments as needed and removes them from memory to make space for additional segments. Thus, the total size of a program can be larger than total available memory. The only constraint is that memory must be large enough to contain calling and called segments simultaneously. (A calling segment is one that calls a subroutine in a different segment.)

Code segments are created by the compiler. Each succesful run of the compiler creates a code file containing at least one segment. When the compiler produces a program with only one segment, this segment contains the code of the program itself (often referred to as the "outer level") as well as the code of all functions and subroutines within the program.

The programmer may, however, cause the compiler to place any function or procedure in a separate segment simply by preceding the reserved word PROCEDURE or FUNCTION with the reserved word SEGMENT. In that case the compiler will produce a code file with one segment for the program and its non-segmented functions and procedures, and a separate segment for each function and procedure preceded by the word SEGMENT. (Segmented functions and procedures may occur at any level. A segmented function or procedure may itself contain segmented or unsegmented functions or procedures.) The "outer level" segment is called the **primary** or **principal segment** and has a special significance, as will be seen later.

Although the compiler can be instructed to break up the procedures of a program into separate segments as it compiles, it is not necessary for all the segments of a program to be created during the same compilation or to be resident in the same code file. It is possible to gather together a group of functions with a common purpose and compile them into a single code file (for example, a group of screen-handling routines). These routines, called a "unit", can then be used by any application which requires them.

Thus, the code file using the unit (called the "host") is physically separate from the unit and may be created independently of it. When the host is executed the operating system finds the unit, and its segments are treated as if they were segments belonging to the host. The host may freely call routines in the unit (those established as public when the unit was written) and the operating system swaps the unit segments as needed, along with the host's segments.

Of course, the notion of the unit not only saves space, it makes the job of the programmer much easier since the programmer (hopefully) writes as much code as possible in the form of reusable units.

There are, then, two kinds of code files: programs and units. Each has segments of code and both local and global data (see sections 3.1 and 3.2 for a discussion of the mechanics of using units). Structurally, programs and units are close to identical. The primary difference between them is that programs are designed to start things off–their "master" (primary) segments run first. Units are designed to have their segments called upon by hosts. (A unit can be a host to another unit. See also section 5.0.17 for a means of creating a unit that, like a program, can contain an initially running segment.)

This scheme imposes a bookkeeping burden on the operating system, since, before execution can begin, pieces of the program must be located in various different code files. Therefore each code file contains embedded information to facilitate the association of the various segments required by a program. And the operating system maintains data structures enabling it to track these segments.

The term **compilation unit** designates program source suitable for submission to the compiler. In Pascal, a compilation unit is generally a program or a unit. As mentioned, each can contain one or more segments.

The compiler, when translating a compilation unit, distinguishes between segments by assigning each segment a **segment number.** These numbers are assigned to segments contained within the compilation unit itself and to segments "used" from other compilation units. When a procedure from a different segment is called, the compiler generates code containing the segment number of the segment containing the called procedure.

Segment numbers are not permanently associated with segments. They are applicable only within a compilation unit. Thus, if program PROG1 uses a segment SOMESEG from unit UNIT2, SOMESEG may be assigned a segment number of 6, for example when PROG1 is compiled. But if a different program, PROG2, uses SOMESEG, that segment may very well be assigned an entirely different segment number when PROG2 is compiled. Segments *do* have permanent identification in the form of an eight-character segment name.

Segment information is maintained in two places within a code file. For segments local to a compilation unit, segment information is contained in the **Segment Dictionary.** This contains the segment number, segment name, and many other pieces of information needed by the operating system when it constructs the environment of a program prior to its execution. The Segment Dictionary is discussed in section 8.4.2.1.

The compiler does not have the Segment Dictionary information available for segments "used" from other compilation units. This information is available in the code file containing the used segment, which has its own Segment Dictionary. However, the operating system must have this information when the program is being prepared for execution. Therefore the compiler constructs a list of all "foreign" segments with their names and segment numbers. The names come from the USES statement which must appear in any Pascal compilation unit that uses segments from other compilation units. The numbers, as mentioned, are assigned by the compiler.

This list of "foreign" segments is called the **Segment Reference List,** and is stored in the code file of any compilation unit using foreign segments. The Segment Reference List is discussed in section 8.4.2.3.

Prior to the execution of a program the operating system traverses the Segment Reference List and searches for the segments named therein in SYSTEM.LIBRARY (or other code files contained in USERLIB.TEXT). Once the segment is discovered all the information required by the operating system is available in the Segment Dictionary of the used segment's code file.

The process of locating the segments of a program prior to its execution is referred to as the construction of the program's environment. The time that this takes place is called **associate time.**

The operating system uses two important data structures when constructing a program's environment. These are the **Segment Information Block (SIB)** and the **Environment Record (E_REC).**

The operating system constructs a SIB for each segment – foreign or local – that may be used by the currently executing program. The SIB contains two kinds of information. First, the SIB contains information regarding the size of the segment and its location on disk. Remember that not all segments of the currently executing program need be resident at once. Segments are dynamically swapped into and out of memory as space dictates. Therefore the operating system must always know where to find each segment on disk. The operating system fills the SIB fields for a segment's size and disk location from the Segment Dictionary of the code file containing the segment.

Each SIB also contains information about a segment's current memory location (if it is resident) and activity. The operating system uses this information to determine which segments may be swapped out of memory when space is required, and which segments may be removed from memory entirely. SIBs are discussed in section 8.4.3.

A segment may reference other segments and it may itself by referenced. To distinguish between these two roles in the discussion that follows a segment will be described as a either a "referencer" or a "referencee".

Recall that the compiler assigns local segment numbers to all referencees. These referencees are located, and SIBs for them are constructed, by the operating system at associate time using the Segment Dictionary (for referencees in the same code file) or the Segment Reference List (for foreign referencees). But calls to referencee segments during execution are by number alone. There must be a means of relating a referencee's local number to its SIB during execution, so that the segment can be located in memory or loaded from disk if it is not resident. Therefore the operating system maintains an Environment Record for each *referencer* segment - foreign or local - that may be used by the currently executing program (since all segments are potential referencers -any segment may call another -this means that every segment has an E_REC).

Each referencer's E_REC contains (among other things) a pointer to the referencer's SIB and an array of pointers to the E_RECS of all the referencee segments (foreign and local) called by the referencer. When, during the execution of the code of one segment, a reference is made to another segment, the called (referencee) segment's number is used as an index into the current (referencer) segment's array of pointers. The E_REC, and thereby the SIB, of the called segment can then be located. Note that since the same set of segment numbers is used throughout an entire compilation unit, all the local segments of a compilation unit will have identical arrays of pointers to E_RECs. E_RECs are also discussed in section 8.4.3.

8.4.2 Code File Structure

Although a code file is a file like any other as far as the file system is concerned, it does have a more complex structure than most other files. A code file may contain the following distinct entities: A Segment Dictionary, a Segment Reference List, INTERFACE text and one or more code segments.

As its name implies, the Segment Dictionary contains detailed information about the segments within a code file. There may be up to 256 segments in a code file, and the size of the Segment Dictionary depends on how many segments there actually are. The Segment Dictionary is not stored contiguously but is broken into block-sized chunks (remember that a block is 512 bytes).

Each block contains information for up to 16 segments. The first block of a code file always contains the first block of the Segment Dictionary. If there are more than 16 segments in the code file, additional blocks are allocated to the Segment Dictionary, but these may appear anywhere within the code file. The blocks in the Segment Dictionary take the form of a linked list, as will be discussed in more detail in the next section.

The Segment Reference List is an array of records, one for every "foreign" segment referenced in this compilation unit. Each record contains a "foreign" segment's name, and the number by which it is referenced in this code file. The Segment Reference List is stored immediately following the code of the file's principal segment. It need not begin on a block boundary.

In order for a program to use the data and procedures of a separately compiled unit, the programmer of the unit must establish an INTERFACE section wherein the "public" (i.e., available to USEing hosts) entities are declared. When the host is compiled the compiler "edits" the unit's INTERFACE section into the text of the host and the entities in the INTERFACE section are available in the host as if they were declared as global within the host itself.

The INTERFACE section of a unit is stored within the unit's codefile. It is stored in the form of text (following most of the conventions for p-System text files; see section 8.0). The INTERFACE section takes up an integral number of blocks but it need not occupy an even number of blocks as is the case with an ordinary textfile. It may appear anywhere within a code file; its location is pinpointed in the Segment Dictionary.

Of course, code files also contain segments of code! These segments may appear anywhere within the code file. They must begin on a block boundary but may be of any size. Their locations and sizes are specified in the Segment Dictionary. Code segments themselves have a fairly complex structure which is discussed in section 8.4.2.2.

Note that the requirement that segments begin on block boundaries, together with the fact that they may be any number of words long, implies that there may be unused "filler" within a code file.

8.4.2.1 Segment Dictionary

WARNING: It is not the intent of this section to present an exhaustive description of the Segment Dictionary data structure. The reader interested in such a description is referred to the Internal Architecture Guide or equivalent document. Our purpose here is to describe the Segment

Dictionary in relatively general terms and thereby facilitate comprehension of the IAG's detailed discussion.

The Segment Dictionary is the source of most of the information retained internally by the system for a segment during the execution of a program. As mentioned previously the Segment Dictionary occupies the first block of a code file and as many additional blocks as are necessary to describe all the segments of the file. Each block describes up to 16 segments. The information in each block is arranged in the form of six arrays, with the information for a particular segment in corresponding positions in each of the arrays.

The first array gives the block number, relative to the beginning of the code file, of the beginning of the segment. It also contains the size of the segment in words. Principal Segments contain a Segment Reference List; however the size of the Segment Reference List is not included as part of the size of the segment.

The next array contains the name of the segment. The following array contains the segment type - it specifies whether the segment is a primary segment, and if so, whether it is a program or unit. It specifies if the segment is a native code segment. Also in the same array is an indication as to whether the segment needs to be linked. Generally, only Assembler routines and procedures which use EXTERNAL routines need to be linked. Finally, this array specifies whether the segment is relocatable - that is, can the segment be moved by the system once it is loaded into memory, or must it be locked into the position it originally occupied.

The array which follows contains the block number within the code file of the INTERFACE text for the segment. This information is applicable only to the principal segment of a unit.

The fifth array contains three pieces of information: the local segment number (as assigned by the compiler), an indication as to whether there is processor-specific code in the segment and the p-System version under which the segment was compiled.

The last array contains different information for principal segments and for subsidiary segments. For principal segments this array specifies the size of the global data for the code file, the size of the Segment Reference List (in words), the highest local segment number used in the code file (which therefore includes "foreign" segments, since they are also assigned local segment numbers) and the size of the INTERFACE text (the block number of the INTERFACE text was supplied in a previous array).

For subsidiary segments the last array specifies the name of that code file's principal segment.

Aside from the six arrays each block of the Segment Dictionary also contains a pointer to the next block of the Segment Dictionary (its block number within the code file), a copyright notice and a single word containing the value 1, as an indication of byte-sex.

The issue of byte sex deserves further explanation at this point.

On most mini and microcomputers the word (two bytes) is the basic unit of storage - although these machines are byte-addressable, most data occupies an integral number of words. Under the p-System too, most data items occupy an integral number of words.

A word has a Most Significant Byte and a Least Significant Byte. The MSB is considered to be the byte containing the "high-order" portion of the data. On some machines the MSB is the one with the lower address of the two bytes in a word. For example, if a word occupies bytes 0 and 1 the MSB would occupy byte 0 on these machines. The Motorola 68000 is an example of such a machine.

But on other computers the LSB is stored in the lower address. If a word occupies bytes 0 and 1, byte 0 would contain the LSB. The DEC PDP-11 series is an example of this sort of computer.

This distinction is known as byte-sex. Although p-code is interpreted and therefore transportable between machines, code files moved to a processor with a different byte-sex must be "byte-flipped" before they will execute properly. The high-order and low-order bytes of each word of code must be switched.

This raises the question, how does the interpreter know which type of machine the code originated on? How does it know whether or not to "byte-flip" the code before attempting to execute it?

To enable the interpreter to know whether a code file is of the "local" byte sex or the opposite, a specific word is selected within a code file and the value 1 is stored therein during compilation.

The interpreter examines this word before attempting to execute the code file. If the file is being executed on a machine of the same byte sex as the one upon which it was compiled, the word will contain a 1 and no "byte-flipping" is necessary. If the code is executed on a machine with the opposite byte sex, however, the value of the word will appear to be 256 rather than 1! This is because the two bytes of the word will appear reversed to the interpreter; rather than seeing 0000 0000 0000 0001 the interpreter will see 0000 0001 0000 0000, or 256. In that case the interpreter will know that the code file originated on a computer of the opposite byte sex and will flip each word of code before attempting to execute it.

8.4.2.2 Segment Format

A p-System segment, as mentioned previously, is a code "atom". We can examine its various parts, and will do so in this section. However, the system does not manipulate code smaller than a segment. Thus, loading of code from disk to memory, unloading of code from memory and movement of code in memory to make room for additional code all take place upon integral segments.

Naturally, the essential ingredient of a segment is executable object code. This executable code is divided into procedures, corresponding to the high-level language procedures that spawned them. In addition to its code, each procedure contains information needed by the system when the procedure is executed. The segment contains a dictionary that enables each procedure within that segment to be easily located. This is called the **Procedure Dictionary.** A segment may contain up to 255 procedures and thus there are up to 255 entries in the Procedure Dictionary.

In addition to procedure-specific information a segment also contains general information pertinent to the segment as a whole.

One entity that segments may posess is a **Relocation List.** The Relocation List may be used when a segment contains native code in one or more of its procedures. When native code procedures are present it is possible that address references may be in terms of absolute memory locations. This implies that the code will run only when loaded into specific and precise memory areas. However, the operating system demands the right to determine where to place a segment in memory at the time the program is run. Therefore a method must be provided to enable the operating system to convert the absolute memory references in the code to correspond to the area where the program will actually be placed.

The Relocation List is the mechanism provided by the p-System to permit the adjustment of absolute memory references. A detailed description of the Relocation List is beyond the scope of this discussion. Briefly, it contains pointers to those objects that require relocation (i.e., modification of absolute memory references) together with a specification of the type of relocation necessary. Memory references in p-code are relative rather than absolute so no relocation information is necessary.

A segment begins with a pointer to the Procedure Dictionary, followed by a pointer to the Relocation List. The Relocation List is the very last thing in the segment since it is not always needed during execution and may be discarded. The Procedure Dictionary immediately precedes the Relocation List.

The Procedure Dictionary begins at a high address and procedes downwards to lower locations. It begins (at the high address) with a count of the number of procedures in the segment. There are <count> number of words beyond in the Procedure Dictionary. Each is a pointer, relative to the beginning of the segment, to the code for a procedure.

In a high-level language procedures are called by name. But the compiler generates code that references procedures by procedure number. During execution a procedure call is accomplished by locating the procedure via the Procedure Dictionary. The procedure number in the range 0-255 is simply the index into the Procedure Dictionary.

The pointers to the Procedure Dictionary and Relocation List are followed by the name of the segment. (Segment numbers, remember, are local to the compilation unit and may change as segments are used by different hosts. Thus, segment numbers are not stored as part of the segment itself but rather in the compilation unit's Segment Reference List. Segment names are fixed; they do not vary from one host to another. Thus, they may be recorded within the segment itself.)

Each segment then contains a word with value 1 to indicate byte sex. The issue of byte sex is discussed in the previous section.

The next word in the segment is a pointer to the segment's **Constant Pool.** Constants are simply values embedded within code. Under Version IV of the p-System single-word constants are placed in the procedure code itself. But all constants larger than one word within a segment are grouped together into a common area called the Constant Pool. The Constant Pool is located directly before the Procedure Dictionary. (When a code file has no multi-word constants the value of the pointer to the Constant Pool is zero.)

When the compiler encounters a reference to a constant it generates code that accesses the constant in terms of its distance into the Constant Pool. Thus, constants are located during execution by finding the start of the Constant Pool via its pointer, then selecting the desired constant based on its offset into the pool as it appears in the code.

Constants that have real values are distinguished from other constants under the p-System. Real values are represented, and real arithmetic performed, in a processor-dependent fashion. The compiler, which must generate code that will run on *any* processor, cannot therefore generate real constants in their final form. Instead, the compiler represents real constants in a special, machine-independent form. Real constants must be converted to the internal representation of the processor in use before execution.

Real constants are therefore stored in a separate sub-pool within the Constant Pool. This sub-pool may be anywhere within the constant pool. The Constant Pool begins with a pointer to the real sub-pool. The pointer is relative to the beginning of the Constant Pool itself and has a value of zero when there are no real constants. Thus, when a segment is loaded, the system can locate the real constants and convert them to the current processor's specific representation.

NOTE: The process of converting real constants to processor-specific form when a segment is loaded can be a time-consuming operation. Therefore a utility (REALCONV) is provided to convert real constants and store them in place in converted form within the Constant Pool. Effectively, the real sub-pool is merged with the main Constant Pool and the sub-pool pointer is set to zero to indicate to the system that no conversion need be performed at runtime. Code files converted in this manner are no longer portable to machines with different processors, however.

The pointer to the Constant Pool is followed by a word indicating size of real values for this segment. Version IV of the p-System provides support for two- and four-word real values. The compiler is shipped to generate one of these by default (usually four-word reals). The alternate size can be generated using the {$R4} or {$R2} compiler directives (section 5.0.14).

Following two unused words each segment contains the object code for each of its procedures.

The object code for each procedure begins with a word called **DATASIZE.** When a procedure is called, its local variable are allocated space on the stack (which is freed upon exit from the procedure). The number of words to be allocated for a procedure is indicated by its DATASIZE.

DATASIZE is a positive value for procedures beginning with p-code. To flag procedures beginning with native code, the DATASIZE value is one's-complemented (bits are changed to their opposite values). For outer level routines in principal segments DATASIZE is zero for reasons explained in the following paragraphs.

The outer level of a program or unit in a principal segment may also have "local" variables, but since it *is* at the outer level these are really global and accessible to the entire program. As such, they are allocated and available during the entire life of the program.

There are separate instructions in p-code for accessing global and for accessing local variables. The "local" variables of the outer level routine in a principal segment may be viewed as either local or global *when within that segment*. But the compiler always generates code treating the variables as global under such circumstances. Now, if the DATASIZE for the outer level routine were some positive value, stack space for local variables would be allocated when that procedure was entered. That space would never be used, however, since outer level variables are accessed as global and stored as global on the stack. (Recall that one of the entries in the Segment Dictionary for outer segments specified amount of stack space for global variables.) The wasted space would survive the entire life of the program since it was allocated for the outer level routine, which is exited only when the program terminates.

Therefore the DATASIZE value for an outer level routine in a principal segment is zero.

Following the DATASIZE word at the start of a segment is the **EXITIC** word.. This is a pointer, relative to the beginning of the segment, to the code to be executed upon exit from the routine. EXITIC is followed by the object code itself.

Procedures, with their accompanying DATASIZE and EXITIC values, follow each other without break in the segment.

8.4.2.3 Segment Reference List

The compiler generates references to both local and foreign segments in terms of segment number. When the operating system constructs the environment of a program prior to its execution, information regarding the local segments is available in the code file's own Segment Dictionary. The segment number can be located therein and the necessary information for that segment can then be obtained.

However, no information is available in the host code file for foreign segments. Therefore, to locate these segments prior to the execution of a program a Segment Reference List is maintained for each program or unit using foreign segments.

The Segment Reference List contains the names and the segment numbers of all the foreign segments used by the current compilation unit. If there are no foreign segments there is no Segment Reference List. When present it is located immediately following the principal segment, after its Relocation List. The Segment Reference List is never needed once all the segments of a program are located, and its position at the end of the segment makes it easy to discard once the system is done with it. It also makes it easy to locate. The Segment Dictionary can be used to find the beginning of the principal segment; the length of the principal segment (also to be found in the Segment Dictionary) can be used as an offset to the Segment Reference List.

The Segment Reference List takes the form of an array with one element for each foreign segment. The elements take the form of records, with a field for segment name, a field for segment number and some filler. The total number of elements in the array is given in the Segment Dictionary.

When the operating system constructs the execution environment of a program it traverses the Segment Reference List element by element, attempting to find each segment. It searches the system library file and then the code files named in the user library directory file (by default, USERLIB.TEXT), checking the Segment Dictionaries of each of these files in turn until it finds a match for each segment named in the Segment Reference List. Once a segment is located a SIB and E_REC are constructed for it (if these do not already exist;see section 8.4.4).

NOTE: The search for segment information using the Segment Reference List can take a long time, especially for programs that use segments from many units. This is because the operating system must reread from disk the Segment Directory of SYSTEM.LIBRARY and all the code files named in USERLIB.TEXT for *each* segment named in the host's Segment Reference List until the segment is found. For programs sold as complete entities the Librarian utility can be used to bind all the necessary segments together into a single code file. For development purposes, however, this is inadequate. Tying the segments together into a single code file makes for a large code file. When the segments are general-purpose and used by many applications (one of the goals of the segment scheme) they will have to be duplicated in each application which uses them, wasting large amounts of disk space. Versions of the p-System distributed as this is written include a utility called QUICKSTART which, in effect, causes the Segment Dictionary information of *foreign* segments to be included in the *host's* code file. Thus, the code file remains essentially the same size, the "used" segments remain separate and only one copy need be retained, but the time needed to build the environment of a program is significantly reduced.

NOTE: If there are multiple segments with the same name in different code files searched, the first encountered is the one used. If a segment named in the Segment Reference List is not found an error message is displayed and the program is not run.

8.4.3 Environment Records & Segment Information

Much of the information maintained regarding segments in the Segment Dictionary is necessary not only when the environment for a program is constructed but also throughout the entire life of the program - from the moment it begins execution until it finally completes. This is true not only for the segments in the host compilation unit but for all segments, regardless of unit of origin.

The p-System's dynamic memory management scheme causes segments to be swapped into memory as needed, and out of memory to accommodate additional segments. Thus the disk address, for example,of a segment (recorded in its code file Segment Dictionary) must also be resident in memory and available to the operating system as long as the segment may still be needed.

The operating system therefore must maintain data structures for each segment a currently executing program may use. These data structures must record the Segment Dictionary information pertinent to that segment.

Other information must be recorded internally regarding segments of currently executing programs. The operating system must keep track of each segment's activity relative to the other segments so that it can intelligently determine which segments to swap out when additional memory space is needed. The operating system must keep track of a segment's current memory location. This value may change as the program runs; a procedure call into a segment requires, obviously, that the current address of the code be available.

This sort of segment information is maintained in a data structure called a Segment Information Block, or SIB. There is one SIB for each and every segment that may be used by a currently executing program. The SIB is allocated on the heap when the environment of a program is constructed. Its values come from the Segment Dictionary of the code file containing the segment. The SIBs remain on the heap until the program completes execution even though individual segments may be swapped out of memory for part of that time.

Each SIB begins with a pointer to its segment's current memory location. The value of this pointer is NIL (an integer with machine-dependent value) for a non-memory-resident segment. The pointer is loaded with the initial address of the segment at the time the segment is brought into memory. It is updated by the operating system whenever the segment is moved.

NOTE: Many p-System implementations maintain separate logical memory spaces for code and for data, thus utilizing more than 64kb of memory. Under such implementations the pointer to a segment's memory location is not absolute but must be further resolved using the SYSTEM.MISCINFO code pool resolution field.

Following the pointer to the segment's memory base are three fields that help the operating system determine the relative usage of the segment. These include a count of the number of yet-to-be-completed calls to procedures in the segment from procedures outside it, an "activity" counter (value determined by a formula beyond the range of this discussion) and a count of the number of E__RECs pointing to the SIB. In effect, this value indicates how many other segments may use this one; when the value is zero (typically at the end of the program) the segment and its SIB are no longer required; the SIB may be removed from the heap. E__RECs and their pointers will be further discussed shortly.

Each SIB has a flag (in the form of an integer) which indicates its segment's relocatability status. A segment may be position locked; that is, it must always occupy its original memory locations. It may be be actually swappable or it may only be potentially swappable. A potentially swappable segment is one which has explicitly been memory-locked by a program using the MEMLOCK intrinsic. The MEMSWAP intrinsic can be used to render the segment actually swappable again (see sections 4.20 and 4.21).

Each SIB also contains the following information which is derived from the Segment Dictionary: Segment name, size, disk address and volume information (for the drive and volume containing the segment).

SIBs of principal segments contain the size of the global data for that compilation unit. As we shall see, the global data itself is pointed-to from within the E__REC.

The SIBs for all active segments are organized in the form of a doubly-linked list. One end of the linked list is pointed to by the KERNEL variable `CODEPOOL.POOLHEAD`, the other end by `CODEPOOL.PERMSIB`. Using either of these as its starting point, the system (or any program) can traverse all the SIBs since each SIB has a pointer to its predecessor and successor.

Recall that the compiler generates references to other segments used by a compilation unit in terms of local segment numbers. These segment numbers are used both for local and foreign segments. Now, imagine a memory-resident segment in the process of execution. A call to a procedure in another segment is encountered in the form of a segment number. The calling segment must locate the called segment for the transfer of control to take place.

The called segment may be in memory or it may be on disk (either having never been loaded or having been swapped out). It is certain, though, that the called segment has a SIB which *is* in memory since all the segments or a currently executing program must have SIBs. The SIB of the called segment will determine whether the called segment is memory resident or not; it will have the memory location of the called segment if it is resident and its disk address if it is not.

The calling segment, then, needs to locate the SIB of the called segment using the called segment's number.

Remember, though, that segment numbers are not absolute; they may change from one compilation unit to another. The called segment may be known as segment 6 to this particular calling segment. But it may be known as segment 18 to a calling segment from a different compilation unit than our calling segment. Yet both calling segments will need to locate the same SIB!

Therefore the association or "mapping" from segment number to the SIB of the called segment must be accomplished individually for each *calling* segment.

When the environment of a program is constructed prior to its execution each segment is given not only a SIB but also an E_REC. The major purpose of the E_REC is to facilitate the mapping between local segment numbers and the SIBs of the actual segments being called.

Every segment's E_REC contains, first of all, a pointer to that segment's own SIB. It then contains an array of pointers to the SIBs of all the segments it may call. The array is indexed using local segment numbers. Thus, when the environment of a program is constructed, each segment's E_REC has its array of pointers (called an E_VECT, by the way) set to the SIBs of the segments it may call; the array is ordered by local segment number.

Note that all the segments in the same compilation unit have the identical array of pointers. The local segment numbers are the same for all the segments in a compilation unit, so when two segments from the same compilation unit call segment 6, for example, they must be referring to the same called segment. But if a segment from a different compilation unit refers to segment 6 it might very well be an entirely different segment. Since this calling segment comes from a different compilation unit its local segment numbers will be entirely different. It will have a different array of pointers.

E_RECs contain some additional information as well. They contain a count of the number of local segments this segment can access; in effect this is an upper bound for the aforementioned array of pointers. They contain a pointer to the global data of the principal segment of the compilation unit containing this segment. E_RECs of principal segments contain a count of the number of *other* compilation units for this program that are using this compilation unit.

Finally, E_RECs of principal segments are tied together in the form of a linked list, so each such segment has a pointer to the next E_REC in the list. The beginning of this list is pointed to by the KERNEL variable `UNITLIST`.

8.4.4 As a Program Runs

Memory under the p-System is shared by a number of entities. Since the p-machine is emulated on most hardware, part of memory is occupied by the interpreter. The remaining memory is shared by the stack, the heap and the codepool.

The stack is used for program and procedure variables. These are allocated as procedures are entered and freed as procedures terminate. The stack is also used to maintain bookkeeping information regarding procedure invocation (activation records). Generally, the stack begins at high memory and grows downwards toward low memory.

The heap is used for dynamically allocated variables. A program or the system may use one of the available intrinsics (such as NEW, for example) to claim space on the heap. This space is explicitly released by the program when it is no longer required.

The p-System uses the heap for its program management records, such as the SIBs and E_RECs discussed in the previous section. The system also maintains disk directories on the heap.

Most code is contained in the codepool. But native machine language routines that are position-dependent are located on the heap. (As we shall see, the codepool is moved around in memory.)

The heap generally begins in low memory and grows upwards.

The stack and heap, then, grow towards each other. But there is normally a large free space between the stack and the heap. This space is used for executable code in the form of segments, as discussed in the previous section. The memory-resident code between the stack and the heap is referred to as the codepool. As a program executes there is constant contention for memory space between the stack, the heap and the codepool.

Those p-System implementations which use extended memory (memory beyond 64kb) do so by partitioning memory into separate areas for code and data. Thus, the stack and heap occupy a partition to themselves, and the codepool occupies a partition to itself. The p-System cannot currently visualize extended memory as a single contiguous area since addresses are maintained as 16 bit values. Addressability is thus limited to 64kb. Partitioning allows the p-System to have two banks of 64kb for a maximum total of 128kb.

When there are separate code and data spaces there is no contention between the stack/heap and codepool. Programs may utilize larger amounts of data, and more code segments may be loaded before swapping occurs.

The codepool always occupies a contiguous area of memory. There is never a gap between segments.

The operating system maintains the allowable bounds of the codepool. These lie between the top of the heap and the "top" of the stack (when the stack is present in the same memory partition; remember that the stack grows downwards). Both the stack and the heap are allowed to grow until they reach the *current* boundaries of the codepool. When the stack or heap attempts to grow beyond the boundaries of the codepool a stack or heap fault occurs. The system will attempt to shove the entire codepool in the opposite direction to make room for the stack or heap. If there is no room to move the codepool (it is already adjacent to the boundary at the other end) the system attempts to remove segments from the codepool. The remaining segments are then moved together and the system again attempts to move the codepool to accommodate the stack or heap. If the system ultimately fails to find sufficient room it crashes with a stack overflow.

When a segment not currently resident in the codepool is called, an attempt is made to read the segment into memory at either end of the codepool without moving the remainder of the codepool. If this is not possible the operating system attempts to shove the entire codepool toward either the stack or heap to provide room for the new segment at the opposite end. If this maneuver fails to provide adequate room for the new segment then currently resident segments are swapped out, the remaining segments are moved together and the system again attempts to move the codepool to accommodate the new segment. As before, if the system ultimately fails to find sufficient room it crashes with a stack overflow.

8.5 File Information Blocks (FIBs)

Pascal's file mechanism provides a machine and operating system-independent means of accessing I/O devices. However, operating systems differ greatly in how they implement I/O. Declaring and using a file in a Pascal program, therefore, causes compilers to generate code that varies widely from one operating system to another.

The p-System uses a data structure called a File Information Block (or FIB) to retain information about a program's files. The FIB declaration appears in the operating system's KERNEL unit; it is printed in the Internal Architecture Guide. Each file declared in a program has an associated FIB.

The FIB contains an item called `fwindow`, which is a pointer to the file's window (the memory location used as destination for a GET or source for a PUT). In addition, the FIB contains the size in bytes of the records in the file, the volume containing the file, a copy of the file's directory entry, various flags that indicate the file's current state, a flag that reflects whether or not the file has been modified and other information.

When the Pascal compiler encounters a file declaration it allocates space for a FIB on the stack, just as it allocates space for a data item of any type. The compiler allocates space for the file's window variable immediately following the FIB on the stack. In effect then, a Pascal file is nothing more than a FIB and a window variable; references to a file are references to the FIB and window variable allocated by the compiler for that file.

Pascal programs normally refer to files using intrinsics such as RESET, REWRITE, CLOSE, etc. When the compiler encounters these intrinsics in a program it generates calls to operating system routines that access the FIB and perform the necessary I/O. It is normally unnecessary for a program to explicitly access a FIB.

There are occasions, however, when the ability to explicitly access a FIB can lead to a more efficient program. As an example of such a situation consider a master file update program. Programs of this genre read records from a current file, perform update operations on various fields in the record and write the updated record out to a new file. The flow of such a program typically is to read a record, copy it to the window variable of the output file, update the record, write it to the output file and continue the process for all the input records.

Note that the input and output files each have their own FIBs and their own window variables. That makes it necessary to copy each input record to the output file's window variable before updating and writing it.

If it were possible to have the input file and output file share the same window variable, however, the copy operation could be eliminated. The flow would then be to read a record, update it in place and write it immediately to the output file. The time savings could be considerable, especially for large files or files containing large records.

Pascal normally requires each file to have its own window variable. This window variable is allocated on the stack along with the FIB when the file declaration is encountered; its memory address is placed in the FIB, in the `fwindow` field.

If we allowed our master file update program to access the FIBs of the input and output files, though, it could force both files to share the same window simply by assigning the value of input file's `fwindow` pointer to the output file's `fwindow` pointer. GETs from the input file and PUTs to the output file would access the same memory location.

It is not difficult to allow a program access to a FIB, as is illustrated below:

```
program tst;
uses kernel;
const
  STO = 196;          {this p-code will be used to assign}
type                  {pointers with unlike base types}
```

```
    thing = record  {the sample base type for files}
      item1: integer;
      item2: char;
      GROSSitem: array[0..1000] of integer;
    end;
  var
    infile, outfile: file of thing;
    fpIN, fpOUT: fibp;       {pointers to FIB, from KERNEL}
  begin
    reset(infile,'oldmaster');          {open the files}
    rewrite(outfile,'newmaster');

    pmachine(↑fpIN,↑infile,STO);
                           {force fibps to point to FIBs of}
    pmachine(↑fpOUT,↑outfile,STO);
                           {infile and outfile. This will allow}
                           {us to access these files' FIBs}
                           {directly}
    fpOUT↑.fwindow := fpIN↑.fwindow;
                           {THE KEY STATEMENT! make the window}
                           {variable of outfile identical to the}
                           {window variable of infile}
    while not eof(infile) do
      begin
        {outfile↑ := infile↑; <— we don't need to do this!}
        outfile↑.item1 := outfile↑.item1 + 10;
                           {or whatever; do the record update}
        put(outfile); {no "outfile↑ := infile↑;"! just put}
        get(infile);  {after get - data is already there!}
      end;
    close(outfile,lock);
    close(infile);
  end.
```

The program declares `fpIN` and `fpOUT` to be pointers to FIBs (type FIBP is imported from KERNEL). The files are opened in the usual manner; the PMACHINE intrinsic (see section 4.26) is used to take the address of the input file (remember that the address of a file is simply the address of that file's FIB) and store it in `fpIN`, and to take the address of the output file and store it in `fpOUT`. `fpIN↑` and `fpOUT↑` can be used to refer to the FIBs of the input and output files, respectively.

The output file's window variable is unneeded; we intend to use the input file's window variable for both files. The pointer to the output file's window variable can be discarded; remember that the window is on the stack, not the heap, and thus does not have to be deallocated. The assignment of the input file's window pointer to the output file's window pointer is therefore a simple matter.

The program loops through all the records, updating, writing and reading, but not copying. When the loop terminates, the files are closed.

8.6 Accessing Internal Operating System Procedures

The UCSD Pascal UNIT construct provides a means of accessing procedures in other compilation units (see section 3.2 for a discussion of the UNIT construct). Any procedure declared in the INTERFACE section of a UNIT is accessible to any program using that UNIT.

This presumes that the INTERFACE section of the UNIT is available at the time the host program (the program that wishes to use the UNIT) is compiled. The compiler requires the INTERFACE section of the UNIT to resolve procedure calls to the UNIT's procedures.

The code file that makes up the p-System, SYSTEM.PASCAL, consists of a number of separately compiled UNITs combined using the library utility. In the interest of reducing the size of SYSTEM.PASCAL, however, the INTERFACE sections of these units were removed (this is an option provided by the library utility).

A number of operating systems UNITs were deemed to contain procedures of general interest to the UCSD Pascal programming community. These UNITs (which include KERNEL, SCREENOPS and COMMANDIO) are distributed as separate code files with INTERFACE sections. The other UNITs of the operating system were not deemed to contain code of general utility and thus are not made available with INTERFACE sections.

Most of the procedures in these other UNITs perform operating system-level functions. A number of these procedures are used by non-system-level programs; although the programmer does not invoke them explicitly, the compiler generates calls to them as needed. These procedures function as though they were part of a run-time library. For example, the compiler generates calls to procedures within the operating system STRINGOPS UNIT when a program uses UCSD Pascal's string manipulation routines. The compiler generates calls to the FILEOPS UNIT when a program performs I/O.

It is rare for a program to need to call FILEOPS procedures explicitly. There are times when this ability would be handy, however. Accomplishing a call to a FILEOPS procedure without the FILEOPS INTERFACE section is a challenging task. It requires detective work to discover which procedure to call, as well as the number and type of parameters that procedure expects to find on the stack when it begins execution. (The compiler, of course, knows about the FILEOPS procedures and their parameters; it can generate such a procedure call with no problem.)

It requires some programming trickery–once the procedure and its parameters have been identified–to enable the compiler to accept the call without the FILEOPS INTERFACE section being available.

As an illustration of a situation where it would be handy to be able to invoke an operating system procedure, consider a program that needs to dynamically allocate and dispose of files.

File variables are normally considered to be static, rather than dynamic items. That is, file variables are allocated on the stack when a procedure is entered and removed from the stack when the procedure exits. Programs do not normally allocate files "on the fly" using the NEW intrinsic; they do not normally dispose of them explicitly when they are no longer required using the DISPOSE intrinsic.

Yet such a capability would be quite useful to a large class of programs, such as sort/merge utilities, where the number of files required constantly changes, or cannot be known until the program executes. It would also be useful in programs that require the arrangement of file variables into dynamic data structures, such as linked lists or trees.

To implement dynamic files a program must declare the following types: a base type (i.e. a structure that defines the records in the files), a pointer to the base type, a file of that base type (remember that this is a file type, not a file variable) and a pointer to that file type.

The program requires another type, this being a record containing one each of the two previously declared pointers: a pointer to the base type and a pointer to the file type. In the program called `dynofiles` that follows, this record is called `FDRec` (for File Data Record). If a linked list structure is used to dynamically manage the files, this record should also contain pointers to its own type for linkage purposes.

Finally, the program requires a pointer to this record type. In the program `dynofiles`, below, this pointer type is called `FDRecP`.

Dynamic management of files is accomplished using variables of type `FDRecP`. Whenever a new file is required the program performs a NEW on a variable of type `FDRecP` and initializes a file; when the file is no longer required, the variable is deallocated using DISPOSE. Once a program allocates and initializes a dynamic file it may open and close, read and write from it as with any file.

Recall that when the Pascal compiler encounters a file variable declaration it allocates a File Information Block (or FIB; see section 8.5) and a window variable for that file on the stack. In the situation described, however, the compiler will never encounter a file variable declaration–the files in the program under discussion are allocated dynamically. Therefore the compiler will never allocate FIBs or window variables for this program's files; the program will have to explicitly allocate these items as part of the file initialization process, before the file is opened and used.

The problem that requires a FILEOPS routine involves the process of initializing the FIB once it is allocated. If the program attempts to open a dynamic file, even after a FIB and window variable are allocated, the program will crash. This is because the compiler not only allocates FIBs and window variables, it also generates a call to a FILEOPS routine, for each file variable encountered in a program, that initializes the FIB values.

Since the compiler cannot generate the FILEOPS initialization call for dynamic files, the program must issue the call itself.

Assume a programmer wrote a program similar to `dynafiles`, below, to manipulate dynamic files, but the programmer was not aware of the requirement to explicitly issue the FILEOPS initialization call. The program would fail. How could the programmer discover the existence of the FILEOPS call and learn how to incorporate it into the program?

Here is where the detective work comes into play. The programmer would hopefully realize that a program that manipulates dynamic files has to assume part of the burden normally assumed by the compiler. It is therefore reasonable to guess that the reason for the failure of the program is a result of neglecting part of that burden. So the first step in solving the problem is to discover exactly what the compiler does when it encounters a file variable declaration in a program.

To that end, the programmer writes a test program that does nothing but declare a file, as below:

```
Pascal Compiler IV.13 c6t-4      11/ 4/84          Page   1


   1   2    1:d    1  program nothing;
   2   2    1:d    1  type
   3   2    1:d    1    thing = record
   4   2    1:d    1      item1: integer;
   5   2    1:d    1      item2: char;
   6   2    1:d    1    end;
   7   2    1:d    1  var
   8   2    1:d    1    f: file of thing;
   9   2    1:d  303    after: integer;
  10   2    1:0    0  begin
  11   2     :0    0  end.

End of Compilation.
```

Note that the file is of type `thing`, the same type used in the program under study, `dynafile`. Note also that type `thing` is 4 bytes long, and that the FIB and window variable for file `f` occupy bytes with offsets 0 through 303 in the stack (the variable after is declared after the file solely for the purpose of discovering how much space the compiler allocates for the file).

The last 4 bytes of those allocated for file `f` are for the file's window variable, which follows the FIB.

Once the programmer has seen the data structures allocated by the compiler for the file, the compiler-generated code can be examined. The DECODE utility (see the Program Development Manual) disassembles p-code files; the output of DECODE, when run against the code file corresponding to the listing above, follows:

```
Segment: NOTHING   Procedure:   1
Block:   1  Block offset:  26  Seg offset:    26
Data size:     0  Exit IC:    34
     Offset                                Hex code
     0(000):   LAO          1               8601
     2(002):   LAO        301               86812D
     5(005):   SLDC         2               02
     6(006):   SCXG    FILEOPS    4         7204
  exit code:
     8(008):   LAO          1               8601
    10(00A):   SLDC         0               00
    11(00B):   SCXG    FILEOPS    3         7203
    13(00D):   RPU          0               9600
```

A detailed discussion of the p-machine instruction set is beyond the scope of this book. However, it is instructive to examine the few instructions of the simple program above because they illustrate how the compiler generates the call to the FILEOPS procedure.

The focus is on the instructions with offsets 0 through 6 (the remainder of the procedure consists of exit code -code which is executed before the procedure returns). The FILEOPS call is immediately apparent, as DECODE explicitly names the segments containing external procedures. Referring to the Internal Architecture Guide for the SCXG instruction yields the information that SCXG is a call to an external procedure, and that the procedure is procedure number 4 in FILEOPS. This is the information the programmer has been looking for! A program which does nothing but declare a file causes the compiler to generate a call to FILEOPS procedure number 4. Evidently, that FILEOPS procedure somehow initializes the FIB so that the file can meaningfully be used by the program.

The next step is to figure out the parameters to FILEOPS procedure number 4. Without knowledge of the parameters, the programmer will not be able to explicitly generate the call to that procedure in the `dynafile` program.

If there are parameters to FILEOPS procedure number 4, the compiler must generate code to push the parameters onto the stack before the procedure is called. The programmer therefore examines the DECODEd instructions that precede the SCXG instruction. Sure enough, all three instructions that precede the SCXG push values onto the stack. The first instruction is LAO 1. Again referring to the Internal Architecture Guide, the programmer learns that LAO loads the address of a global variable onto the stack. The first LAO instruction loads the address of the item with global offset 1 onto the stack. Looking back at the compiler listing of the test program, the programmer finds that the FIB of the file `f` is at global offset 1. The first parameter to FILEOPS procedure number 4 is the address of a file's FIB.

The next instruction is LAO 301. The programmer remembers that the compiler allocates a file's window variable immediately following the FIB, and realizes that the window variable is 4 bytes long (type thing occupies 4 bytes). Therefore, 301 is the global offset of the window variable. The second parameter to FILEOPS procedure number 4 is the address of the file's window variable.

The instruction preceding the SCXG is SLDC 2. This instruction simply pushes the value 2 onto the stack. The significance of the value 2 is not immediately apparent. The programmer would probably experiment with files of different "things" to see if and how the parameters changed. Intuition obviously plays an important role here. Sufficient experimentation yields the information that the third parameter to FILEOPS procedure number 4 is the size of the window variable, measured in words.

Evidently then, a call to FILEOPS procedure number 4 with three parameters as discussed above is necessary to somehow initialize a file before it can be used. The programmer may wish to use DECODE on the compiled `dynafile` program to verify that the compiler does not generate the call to FILEOPS procedure number 4 for dynamically declared files.

The call to the FILEOPS procedure must be explicitly coded in the program. Without the INTERFACE section for FILEOPS, however, a trick must be employed to force the compiler into generating the call -after having pushed the appropriate parameters onto the stack.

The trick operates as follows: The programmer writes a UNIT called FILEOPS which contains an INTERFACE section that resembles the operating system's FILEOPS and a skeleton IMPLEMENTATION section with nothing but null procedures. Of course, the programmer does not know what the INTERFACE section of the operating system's FILEOPS looks like. No matter, however -the purpose of this phony INTERFACE section is to be able to write a procedure header for FILEOPS procedure number 4, and to force that header to appear in the phony UNIT numbered as procedure number 4.

Procedure numbers depend on the ordering of procedure headers within programs or units. Procedure number 1 is reserved for initialization code of UNITs (if any) or main bodies of programs. The rest of the procedures are numbered sequentially as their headers are encountered by the compiler. Thus, the programmer must assure that the header corresponding to FILEOPS procedure number 4 appears as the third header in the phony FILEOPS INTERFACE section.

This is easily accomplished by declaring two dummy procedure headers prior to declaring the header corresponding to FILEOPS procedure number 4. Note that the programmer does not have to be aware of the actual FILEOPS procedures corresponding to these dummy procedures. Nor does the programmer have to be aware of the parameters to these procedures, if any. These procedures will never be called; their sole purpose is to force the header corresponding to FILEOPS procedure number 4 to be numbered as procedure number 4 when the compiler translates the phony FILEOPS.

Following the dummy header is the header for the procedure corresponding to the operating system's FILEOPS procedure number 4, which we name `finit`. This procedure will be procedure number 4 in the phony FILEOPS as well. Note that for this procedure the parameters must be supplied, since the `dynafile` program calls `finit`.

```
{$U-}
{compile this unit to PHONYFILE.CODE}

unit FILEOPS;   {MUST be U-; otherwise we won't be allowed
                 to use the name FILEOPS!}
interface
type
  PhantomFIB = file; {for the first finit parameter}

{procedure dummy1 is reserved for a unit's
                    initialization section}
 procedure dummy2;
 procedure dummy3;

procedure finit(var f: PhantomFIB;
                    window: integer;
                    recwords: integer);

implementation

procedure dummy1;  begin end;
procedure dummy2;  begin end;

procedure f_init;
begin
  writeln('Whoops! We shouldn''t be here!');
          {we hope NEVER to execute this; see notes below}
end;
end {FILEOPS}.
```

The first parameter to `finit` is of type `file`. As mentioned, a file is a FIB to the compiler. This parameter is a VAR parameter, which means that the compiler will pass the address of the FIB. As noted from the DECODE output, the first parameter to FILEOPS procedure number 4 (which is called `finit` in the phony FILEOPS) is the address of the FIB of the file to be initialized.

NOTE: Using a file type makes the `finit` procedure totally general –any kind of a file, of any base type, can be passed to to a VAR file parameter. The compiler does not check type compatibility for VAR file parameters.

The second parameter corresponds to a pointer to a file window. Since the type of a file window varies according to the base type of the file, it has been made an integer for purposes of generality. A host program (`dynafile`, in this example) will allocate a window, then pass ORD(the pointer to the window) to convert the pointer to an integer for `finit`. Pointers are represented internally as integers, so once the operating system receives the parameter it may resume treating it as a pointer.

DECODE informed the programmer that the third parameter represents the size of the window variable in words. (In fact, `finit` should be passed 0 for interactive files, -1 for untyped files and -2 for text files.) The size of the window variable in words is 1/2 of the SIZEOF a dynamic file's base type for any base type less than MAXINT bytes long.

The phony FILEOPS unit must be compiled with the U- compiler option (see section 5.0.12); this permits use of the name FILEOPS which is otherwise reserved for system use.

The IMPLEMENTATION section of the phony FILEOPS consists of null procedure bodies for each of the procedure headers appearing in the INTERFACE section. The procedure body for `finit` could also have been made null; it consists instead of the WRITELN to highlight the fact that, although the `dynafile` program will call `finit`, the body of `finit` in the phony FILEOPS will never be executed!

What happens instead is this: When `dynafile` (or any host program using the phony FILEOPS and calling `finit`) is compiled, the compiler generates a SCXG instruction exactly like the one in the small test program. The segment named is FILEOPS, since our phony UNIT also has that name. The procedure is number 4, since the programmer took care to place `finit` in that position in the phony FILEOPS. And the parameters passed to `finit` when it is invoked in `dynafile` are pushed onto the stack exactly as they are pushed when the compiler itself generates the call; the programmer took care to declare `finit` so that its parameters matched those of the "real" FILEOPS.

When the program using the phony FILEOPS is executed, however, the operating system will *not* haul in the phony FILEOPS segment and try to execute *its* procedure number 4. When constructing the runtime environment of a program the operating system always searches for segments internally (in SYSTEM.PASCAL) before looking elsewhere (see section 8.4.4 for more information on the construction of a program's runtime environment). Therefore the call to FILEOPS procedure number 4 generated by the compiler in the host program will cause the actual

FILEOPS procedure number 4 to be executed! And it will execute correctly, since the correct parameters were passed to it. The dynamic file whose FIB and window variable information were passed to `finit` will be properly initialized by the operating system, and the file can be opened and used without error.

```
program dynafiles;
uses {$U phonyfile.code} fileops;
type
  ThingP = ↑things;       {pointer to the base type}
  things = record         {the base type in this example}
    item1: integer;
    item2: char;
  end;
  phyleP = ↑phyle;        {pointers to files must be declared
                           BEFORE the file itself!}
  phyle = file of things;
                   {the file type which we will be working
                    with dynamically}

  FDRecP = ↑FDRec;        {pointer to the linked list record}
                          {(or File Data Record — FDRec)    }
  FDRec = record          {the linked list record, which
                               consists of..}
    f: phyleP;            {..a pointer to a file of our
                               base type}
    wndow: ThingP;        {..a pointer to a window variable
                             for the file}
    NextFile: FDRecP;     {..a pointer to the next element
                             in the list}
  end;
var
  first,latest,next: FDRecP; {for linked list management}
  int: integer;              {used to generate test data
                              for our example}
  ch: char;                  {ditto}
  s: string;                 {for interaction with user
                              in this example}

procedure OpenFile(var TheFile: FDRecP; FileName: string);
begin
  new(TheFile);    {allocate a record for new file data}
  with TheFile↑ do begin
    new(f);        {allocate a FIB for the new file}
    new(wndow);    {allocate a window variable
                    for the new file}

    finit(f↑, ord(wndow), (sizeof(things) div 2));
         {call on the operating system finit procedure
          to initialize FIB information; see text, and
          comments on unit above for more information}
    reset(f↑,FileName)       {now we can open the file}
  end;
end {OpenFile};
```

```
procedure CloseFile(TheFile: FDRecP);
begin
  with TheFile↑ do begin
    close(f↑,lock); {MUST close the file; no default
                     close when exiting a procedure
                     for these dynamic files}
    dispose(wndow); {get rid of the "window"..}
    dispose(f);     {..and the FIB}
  end;
  dispose(TheFile); {get rid of the file data record
                     entirely}
end  {CloseFile};

begin
  int := 1;         {initialize these for our test data}
  ch  := 'A';
  first := NIL;     {begin generating linked list}
  repeat
    write('Next file (<Return> for no more files): ');
    readln(s);
    if length(s) > 0
      then begin
        OpenFile(next,s);
               {allocate a file data record, with FIB and
                window pointer, then open the dynamic file}
       if first = NIL
               {handle linkage to previous element, if any}
          then first := next
          else latest↑.NextFile := next;
        latest := next;
        with latest↑ do begin
          NextFile := NIL;      {this may be the last one!}
          seek(f↑,0);           {do what we want with file;
                                 in this example..}
          f↑↑.item1 := int;     {..we just write a single
                                 record of test data}
          f↑↑.item2 := ch;
          int := succ(int);     {generate new test data
                                 for next file}
          ch := succ(ch);
          put(f↑);              {write the test data to
                                 the file}
        end;  {with}
      end;  {if length(s)...}
  until (length(s) = 0);

  latest := first;
             {let's prove the data really is there by
              going through the linked list and reading
              it back; at the same time, we'll clean up
              after ourselves by closing and deallocating
              the files}
  repeat
    if latest <> NIL
      then begin
        with latest↑ do begin
```

```
              {read in the record from current file and
               and write it out}
          writeln(f↑↑.item1, ' ', f↑↑.item2);
          first := latest;
              {get ready to do the same for the next file}
          latest := NextFile;
        end;
        CloseFile(first);
            {close the dynamic file, then deallocate the
             window record, FIB and file data record}
      end;
    until (latest = NIL);
  end {dynafiles}.
```

This program illustrates how dynamic files may be used to construct a linked list of files. The base type of the files is type thing. A pointer to type thing (ThingP) is declared; this is used to allocate window variables for the dynamic files. A file type is declared with base type of things, and a pointer to the file type (called phyleP) is declared. Note that phyleP is declared prior to type phyle itself. Declaring a pointer to an as-yet-undeclared type is always permitted in Pascal, but in this case it is required; the compiler will not permit a pointer to be declared to an already-declared file type.

A record type called FDRec is declared. Items of this type will make up the linked list of files. Each FDRec contains a pointer to the base type for a file's window variable and a pointer to the file type for the file itself (i.e. the FIB). Since the linked list of files will grow and shrink dynamically, a pointer to FDRec (FDRecP) is declared so that the file data for each file can be allocated as needed.

The program consists of two REPEAT loops. The first constructs a linked list of files, permitting the user to enter as many file names as are required. Each file is opened and a record of test data is written. The second loop traverses the linked list, verifying that the test data was written correctly and deallocating each file in the list as it is encountered.

Allocation and preparation of the file information is performed in the OpenFile procedure. This procedure is passed the name of a file. Using the NEW intrinsic, it allocates a file data record (FDRec) and – using pointers within the new FDRec–allocates a FIB and window variable for the file. OpenFile then calls the operating system procedure finit to initialize the FIB, and opens the file using RESET. Note that RESET refers to the file not as f, but as f↑. This is because f is of type phyleP, a pointer to a file – the file itself is therefore the data item referenced by f –or f↑. OpenFile returns the pointer to the new FDRec in a VAR parameter.

Deallocation of the file information is performed in the CloseFile procedure. This procedure is passed the pointer to the FDRec of the file to be deallocated. CloseFile first closes the file. This is extremely important, since the compiler will not automatically close dynamic files when exiting the procedure wherein the files were declared (as it does with ordinary ”static” files). `CloseFile` then disposes of the window variable, disposes of the FIB and disposes of the `FDRec` itself.

Note that the files' window variable fields are not referenced as `f↑.item` but as `f↑↑.item`. Again, this follows from the fact that `f` is not a file but a pointer to a file. The file itself is `f↑` and the window variable requires a further level of referencing.

The technique described for invoking FILEOPS procedures, though somewhat complicated to explain, is straightforward in use and can be quite handy. This technique can be used to invoke procedures in any UNIT whose INTERFACE section is unavailable. In fact, it will work for invoking any global procedure, even procedures which do not appear in a UNIT's INTERFACE section at all.

8.7 The Compiler/Operating System Interface

Of all the utilities that are a part of the p-System, the compilers –and other language translators, such as the assemblers –are the most integrated into the operating system. Although these programs exist as separate code files, they rely on the operating system to perform certain preliminary operations for them and to perform clean-up operations after they finish execution.

When the C(ompile or A(ssemble options are selected from the main system promptline, the system checks to see if a text workfile is present before the translator is invoked. If not, the system prompts the user to supply both text and code file names. Then it opens the text and code files and calls on the program named SYSTEM.COMPILER or SYSTEM.ASSMBLER.

If no errors are detected by the translator, the system closes the text and code files before the main promptline is re-displayed. If errors are detected by the translator the user is normally given the option to continue the translation, exit the translator, or invoke the editor. If the user opts to invoke the editor, the operating system causes a transfer to SYSTEM.EDITOR; the cursor is positioned at the error and the error message is re-displayed.

NOTE: Because of the heavy interaction between the translators and other system components, it is not normally possible to succesfully CHAIN to a translator program.

This section discusses the mechanics of the translator/operating system interaction from the perspective of the translator program. An example ”toy compiler” illustrates how to write a translator program so that it interfaces properly with the other operating system components.

The first item of concern for the translator is to be able to access the source program text file and the file to which code output is to be sent. If the translator is to be named SYSTEM.COMPILER or SYSTEM.ASSMBLER and invoked from the main system promptline with the C(ompile or A(ssemble options, it cannot open these files. The file names were solicited from the user by the operating system, and the files are already open by the time the translator begins execution.

The translator can locate pointers to the FIBs (see section 8.5 for a discussion of FIBs) of these files in a KERNEL record called USERINFO. The USERINFO record is the primary means of communication between the operating system, translator and editor as they interact in the process of program development. The pointer to the FIB of the text file is called `symfibp`; the pointer to the FIB of the code file is called `codefibp`.

To access the `symfibp↑` and `codefibp↑` files the translator may declare two files of its own called (for example) `tex` and `cod`. Declaring a file causes the compiler to allocate stack space for a FIB for that file. The declared files are never opened, however; instead, the MOVELEFT intrinsic is used to copy the `symfibp` FIB to `tex` (i.e.,to the FIB of `tex`) and the `codefibp` FIB to `cod` (i.e.,to the FIB of `cod`). In that way, the translator may legally access `tex` and `cod` (since they are declared as bona-fide files) yet, since their FIBs are now the FIBs of the source text and output code files already opened by the operating system, the translator will actually be accessing those files.

p-System textfiles are broken into units called pages, where each page is 2 blocks (or 1k bytes) long. It is convenient, therefore, for a translator to have a 2 block buffer to contain successive pages of the text file as the translation process procedes. Since the buffer will contain text, it is best declared as a `packed array[0..1023] of char`. The BLOCKREAD intrinsic can be used to read one page at a time into the buffer, starting with the second page of the text file (the first page of a text file contains header information).

The translator should keep track of the block number at which the current page begins (i.e.,2, 4, 6 etc. –pages will always begin on even block boundaries). This information is required by the editor in the event it is invoked due to an error in the current page. The translator should also keep track of its current position within the page as it traverses the buffer. This number, which is in the range 0..1023 for a 2 block buffer, is also required by the editor if an error is located.

When the translator discovers an error it generates an error number depending on the nature of the error discovered. The translator should display some characters preceding the error, the text of the error and the error number. In addition, the translator should display an error message that verbally describes the nature of the error.

The Pascal compiler accomplishes error message display by storing error messages in a textfile called SYSTEM.SYNTAX. In this file, each line contains an error number followed by a colon, followed by an error message. The Pascal compiler searches through the textfile until it finds the line beginning with the number of the current error. It displays the error message on that line.

Following display of the error information the user should be given the option of continuing the translation process, tranferring to the editor or escaping to the main system prompt line. No special action need be taken if the user chooses to continue the translation process.

If the user chooses to transfer to the editor the translator must store the block number of the current page into the USERINFO field called ERRBLK and the byte offset of the offending character within the page, into the USERINFO field called ERRSYM. In addition, the error number should be stored in the USERINFO field called ERRNUM. The translator should then exit; no special action need be taken by the translator to accomplish the transfer to the editor. The operating system automatically transfers to the editor when SYSTEM.COMPILER or SYSTEM.ASSMBLER leave non-zero values in ERRBLK and ERRSYM. The editor always examines ERRBLK and ERRSYM when it begins execution; if it finds non-zero values it arranges a jump to the appropriate position in the text. If it finds a non-zero value in ERRNUM it looks up the error message in SYSTEM.SYNTAX (as the translator did) and re-displays it.

If the user chooses to escape to the main system promptline the translator must set ERRBLK to zero to avoid a subsequent transfer to the editor. The translator may then exit.

The program below contains a compiler "shell"; the components of a translator that interact with the user and the operating system in the manner described in this section. Program `ToyCompiler` actually translates nothing, however; instead, it scans the input text looking for occurrences of the "!" character. These are treated as "syntax errors" to demonstrate the compiler's handling of such a situation.

```
program ToyCompiler;
uses kernel;
const
  bell = 7;          {ASCII code for bell}
  esc = 27;          {ASCII code for escape}
  illegal = 6;       {the error number in this illustration}
  pagesize = 1023;  {text file page = 2 blocks}
type
  buftype = packed array[0..pagesize] of char;
var
  ch: char;
  s: string;        {contains an error message for display}
  buf: buftype;     {read the source file into here}
  curblk,        {current block in source file}
  bytestosearch, {bytes left in current buffer}
  lbound,        {determines text to display near error}
  dum,i: integer;
  tex,           {the source file}
  cod: file;     {the "code" file, which this toy
                  program doesn't use}

procedure lookup(num: integer; var msg: string);
                {finds an error message in SYSTEM.SYNTAX
                 given an error number}
var
  err: text;    {the file of error messages}
  snum: string; {an error number in string form}
  msgnum,i: integer;
begin
  reset(err,'*system.syntax');
  repeat
    readln(err,msg);
    snum := copy(msg, 1, (pos(':',msg) - 1));
                          {error number of this line}
    msgnum := 0;          {convert snum from string to..}
    for i := 1 to length(snum) do    {integer, in msgnum}
      msgnum := (msgnum * 10) + (ord(snum[i]) - ord('0'));
    if (msgnum = num)     {get rid of the number and colon}
      then delete(msg, 1, (pos(':',msg)));
  until (eof(err) or (msgnum = num));
  close(err);
  if pos(':',msg) <> 0
    then msg := 'Unknown error';
                          {couldn't find error message}
end;

function max(x,y:integer):integer;
begin
  if x > y
    then max := x
    else max := y;
end;

begin
  {a "real" compiler would prompt for a listing
   file at this point}
  writeln;
  writeln('Toy compiler - release level IV.13 c6t-4');
  writeln;  write('One dot per block ');

  with userinfo do begin
    moveleft(symfibp↑,tex,sizeof(tex));
       {get the source file FIB into tex..}
    moveleft(codefibp↑,cod,sizeof(cod));
       {and the "code" file FIB into code..}
  end; {so we can do I/O on the already-open
        files}

  curblk := 2;
```

```
 {the third block of a text file is where the text begins}
  with userinfo do begin
    repeat {until eof(tex)}
      dum := blockread(tex,buf,2,curblk);
           {read in next source page}
      errblk := 0;
           {if it stays 0, system will not chain to editor
            upon completion of compiler}
      errsym := 0;
           {will "travel" to each occurrence of the "!"}
      write('..');
      repeat {until errsym >= bytestosearch}
        bytestosearch := pagesize - errsym;
             {bytes remaining after last "!"}
        errsym := scan(bytestosearch, ='!', buf[errsym])
                                                + errsym;
    {"compile" it - this toy just looks for "!" characters}
        if errsym < bytestosearch          {we found a "!"}
          then begin
            writeln;
            errnum := illegal;   {or whatever error number}
            lookup(errnum,s);    {find the error message}
            lbound := max(0,(errsym - 30));
                    {display preceding 30 characters}
            for i := lbound to errsym do write(buf[i]);
                    {spit out the surrounding text}
            writeln(' <——');
            writeln(s);
            writeln('Block ',curblk,
                    ' at position ',errsym);
            write(chr(bell),
'Type <sp> to continue, <esc> to terminate, or ''e'' to edit'
                                                            );
            repeat read(keyboard,ch)
            until (ch in [' ','e','E',chr(27)]);
            writeln;
            if ch = chr(esc)     {will not chain to editor}
              then exit(program) {since errblk = 0}
            else if (ch in ['e','E'])
              then begin
                errblk := curblk;
                exit(program);   {will procede to editor}
              end;
          end  {if errsym < bytestosearch};
          errsym := succ(errsym);
                   {continue searching this page..}
        until (errsym >= bytestosearch);
                   {until no more "!" found}
      curblk := curblk + 2; {prepare to read next page..}
    until (eof(tex));       {until no more pages in file}
  end  {with};
end  {ToyCompiler}.
```

After displaying some initial messages, program `ToyCompiler` enables the use of the input text and output code files by moving the FIBs of the already-opened files to `tex` and `cod`. Note that in this program `cod` is never used; the MOVELEFT to `cod` is for illustrative purposes only. The variable `curblk` is used to track the block number beginning the current page. `Curblk` is initialized to 2, since the first page of a textfile contains header information necessary to an editor but not to a translator.

The body of the "compiler" consists of a pair of nested REPEAT/UNTIL loops. The outer loop reads in successive pages until the end of the input textfile is reached. ERRBLK and ERRSYM are initialized to zero. ERRSYM is used to contain the byte offset into the page of the latest "!" character. ERRBLK will remain zero until an "!" is found. When a "!" is found ERRBLK is set to the value of `curblk` if the user wishes to transfer to the editor. The inner loop advances ERRSYM further and further into the page until finally no more "!" characters are found. The next page is then read in and the outer loop continues.

When an error is found the preceding thirty characters are displayed, along with the "!" character itself. `ToyCompiler` uses the function `max` to avoid an attempt to print non-existing characters when a "!" is located in the first thirty bytes of a page. The procedure `lookup` is used to search SYSTEM.SYNTAX for an error message corresponding to the error number. `Lookup` searches sequentially; a real translator would presumably use a more efficient search method.

The error number corresponding to ToyCompiler's "error" is always 6, stored in the constant called `illegal`. Of course, a real compiler would generate different error numbers depending on the nature of the error discovered.

Appendix A

STANDARD I/O RESULTS

1. No error
2. Bad Block, Parity error (CRC)
3. Bad Unit Number
4. Bad Mode, Illegal operation
5. Undefined hardware error
6. Lost unit, Unit is no longer on-line
7. Lost file, File is no longer in directory
8. Bad Title, Illegal file name
9. No room, insufficient space
10. No unit, No such volume on line
11. No file, No such file on volume
12. Duplicate file
13. Not closed, attempt to open an open file
14. Not open, attempt to access a closed file
15. Bad format, error in reading real or integer

16. Ring buffer overflow
17. Write Protect; attempted write to protected disk
18. Illegal block number
19. Illegal buffer address
20. Invalid textfile format

Appendix B

STANDARD EXECUTION ERRORS

1. No error
2. Invalid index, value out of range
3. No such procedure in segment
4. Exit from uncalled procedure
5. Stack overflow
6. Integer overflow
7. Divide by zero
8. Invalid memory reference <bus timed out>
9. User Break
10. System I/O error
11. User I/O error
12. Unimplemented instruction
13. Floating point math error
14. String too long
15. Programmed HALT

16. Illegal heap operation
17. User breakpoint
18. Incompatible real number size
19. Set too large
20. Segment too large

Appendix C

CONDITIONS CAUSING I/O ERRORS

1. CRC Error

   Returned whenever a CRC (cyclic redundancy check) or Parity error occurs.

2. Bad Unit Number

   Returned for accesses to a device for which there is no driver declared.

3. Bad Mode

   Returned for attempts to read on a write-only device or write on a read-only device.

4. Undefined Error

   Returned when an error of indeterminable type occurs.

5. Lost Unit

   Returned by the file system only; it indicates that a disk has gone off-line during an I/O operation.

6. Lost File

   Returned by the file system only; it indicates that a file expected to be in a disk directory is not present.

7. Bad Title

   Returned by the file system only; it indicates an attempt to open a file with an invalid file name.

8. No Room

   Returned by the file system only; it indicates either an attempt to open or extend a disk file when disk space is unavailable, or an attempt to open a new file on a disk with a full directory.

9. No Unit/Volume

   Returned either after an attempt to access an off-line unit or after an error occurs during UNITCLEAR. Also returned by the file system to indicate an attempt to access a volume which is not on-line.

10. No File

    Returned by the file system only; it indicates an attempt to open a nonexistent disk file.

11. Duplicate File

    Returned by the file system only; it indicates an attempt to create more than one temporary file with the same file name on a single disk volume.

12. Not Closed

    Returned by the file system only; it indicates an attempt to open a file variable which is already connected to an external file.

13. Not Open

    Returned by the file system only; it indicates an attempt to access a file variable which is not connected to an external file.

14. Bad Format

    Returned by the file system only; it indicates an attempt to read a real value or integer value with incorrect input format.

15. Ring Buffer Overflow

    Returned during a read from a serial device after its input buffer has overflowed. (Not implemented on most systems.)

16. Write Protected Disk

    Returned when attempting to write to a write-protected disk.

17. Illegal Block Number

    Returned when attempting to access a nonexistent block on a block-structured device, or when a seek error occurs.

18. Illegal Buffer Address

    Returned when attempting to initiate an I/O operation with a non-word-aligned starting buffer address. (Applies only to block-structured devices.)

19. Invalid Text File

    Returned by the file system when an attempt is made to create a text file containing fewer than four blocks.

Appendix D

STANDARD I/O UNIT ATTRIBUTES

This section describes the operations defined for UCSD Pascal I/O units in their standard configuration. All I/O operations are performed with the unit I/O intrinsics described in section 3.9.

I/O units can be divided into two classes according to their attributes: **serial units,** and **block-structured units.** A unit's class determines the kinds of operations performed on the unit and the available I/O options. I/O options are specified by setting various bits in the control word parameter of the UNITREAD and UNITWRITE intrinsics.

NOTE: An option is enabled if its bit is set to 1; otherwise, it is disabled. The low-order bit in a control word is bit 0. Unused bits in control words should always be set to 0. For example, a control word value of 6 sets bits 1 & 2 to 1 (and all other bits to 0).

NOTE: The standard UCSD Pascal I/O system may be augmented by user or vendor-supplied custom device drivers; these drivers are not described in this text, but may be described in vendor-supplied documentation.

D.1 Serial Unit Attributes

Serial units read and/or write sequences of characters to a serial device such as a console, printer or remote line. In normal I/O, an input stream or an output stream may include either data characters or control sequences. Input control sequences control the interpretation of input data characters and are not returned as input data. Output control sequences may be expanded to a series of data characters. The control sequences for the console and keyboard devices are a superset of the control sequences for the printer and remote devices. Special treatment of these control sequences may be defeated by using the control options described in the following sections.

D.2 Serial Input Attributes

Each serial input unit maintains its own input queue. All input data characters and control sequences are stored in a device queue before being read and interpreted during an input operation. The exception to this rule is the <break> control sequence, which bypasses the input queue and is interpreted immediately. A device queue size is normally either 1 character or 32 characters, depending on the particular serial device driver installed. See your System Installation Guide for details.

The control sequences recognized by all serial input drivers are:

<alphalock>	Simulates the alphalock key on the keyboard. After receipt of this character, the device driver automatically shifts any upper case alphabetic characters to lower case alphabetic characters and vice-versa. A second receipt of this character toggles the alphalock state off.
<eof>	Is treated as the end-of-file marker; the end-of-file marker is placed in the input buffer and the input operation is terminated immediately.

The control sequences recognized by the keyboard serial input drivers are:

<stop/start>	Alternately suspends and resumes console output.
<eof>	Is treated as the end-of-file marker; a null is placed in the input buffer, the remainder of the input buffer is filled with nulls, and the input operation is terminated.

<flush>	Discards ("flushes") subsequent console output.
<break>	Aborts execution of the current program by causing a **User Break** execution error.

NOTE: Unitread operations on the console (unit 1) are received from the keyboard and echoed to the console. Unitread operations on the keyboard (unit 2) are not echoed. Console output control sequences echoed to the console are subject to the same rules as console output control sequences explicity written using the Unitwrite intrinsic. See the next section for details.

NOTE: Each serial input control sequence may be defined by the user in the System.Miscinfo file (see the System Installation Guide). The normal definitions of these characters are as follows:

Control sequence	Normal definition
<stop/start>	Control-S (13 hex)
<eof>	Control-C (03 hex)
<flush>	Control-F (06 hex)
<break>	Control-@ (00 hex)
<alphalock>	Control-R (12 hex)

The interpretation of serial input control sequences is controlled by the control parameter in the Unitread call. The control options are as follows:

- Bit 2 - Suppress recognition of the <eof> and <alphalock> sequences. Note that if an alphalock condition exists before an input operation is begun, the alphalock condition persists throughout the input operation. Also suppresses DLE expansion when echoing a DLE to the console (see the next section for details).
- Bit 3 - Suppress CR/LF generation when echoing a CR to the console. See the next section for details.

NOTE: It is not possible to suppress <stop/start>, <flush> and <break> sequence interpretation.

WARNING: Although the Setup utility implies that it is possible to specify control sequences preceded by a leadin prefix (such as ESC), no version of the p-System honors this specification. Input control sequences may consist of only one character.

D.3 Serial Output Attributes

The control sequences recognized by all serial output drivers are:

- The ASCII DLE character (10 hex) - treated as the first character of a blank compression character sequence; the next character is defined to contain a byte value which is 32 greater than the number of blank characters to be written to the device. Note that DLE processing applies only to the transmission of text files; it should be suppressed when writing code or data files to a serial device.
- The ASCII CR character (0D hex) - defined as a "new-line" character in text files. Whenever CR is written to a serial device, the I/O system automatically follows it with the ASCII LF character (0A hex). Note that CR/LF processing applies only to the transmission of textfiles; it should be suppressed when writing code or data files to a serial device.

The interpretation of serial output control sequences is controlled by the control parameter in the Unitwrite call. The control options are as follows:

- Bit 2 - Suppress DLE expansion.
- Bit 3 - Suppress automatic generation of a LF after a CR.

NOTE: Serial output is somewhat faster if bits 2 & 3 are set on a Unitwrite call.

D.4 Block-structured Unit Attributes

Block-structured devices read and/or write sequences of characters to a block-structured device such as a disk. All characters involved in a transfer are treated as data; there are no embedded control sequences.

- Bit 1 - Physical sector I/O. Allows access to any physical sector on the disk. Disk sectors are addressed by logical sector number; the first sector on the disk is sector 0. Note that physical sector mode allows normally inaccessible disk sectors to be accessed (e.g., bootstrap sectors if present). The starting block parameter is redefined to denote the starting logical sector number. In this situation, the byte count parameter is ignored and the I/O operation transfers one physical sector of data; the size of a physical sector is determined by the type of the disk currently in use and may be obtained by using the UNITSTATUS intrinsic.

WARNING: Because the byte count is ignored in physical sector mode, a physical sector read may overrun a buffer that is not declared large enough to hold a physical sector. The maximum p-System physical sector size is 512 bytes.

D.5 I/O Unit Specification

This section describes the standard system I/O units. The unit attribute determines the options available for use with the UNITREAD and UNITWRITE intrinsics. (See sections 4.42 and 4.45 for parameter information.) Unit-specific features are described next to the operations affected. The UNITSTATUS record format depends on the type of unit being polled. See section 3.9.3 for details.

NOTE: Users and vendors may supply their own drivers for units 128 through 255. Vendor-supplied documentation may describe the action of these drivers. See the System Installation Guide for further details.

NOTE: Subsidiary volumes and additional serial ports may be assigned to units 13 through 127. These devices follow the general rules for block-structured devices and serial devices, respectively. See the System Installation Guide for further details.

NOTE: On early Version II systems, units 7 and 8 were combined into a bidirectional unit 8 called REMOTE:.

#0:	- Device	- System information
	- Volume Name	- none
	- Attribute	- System
	- UnitClear	-
	- UnitRead	- Halt the system
	- UnitWrite	- Halt the system
	- UnitBusy	- Return FALSE
	- UnitWait	-
	- UnitStatus	- Return system status information
#1:	- Device	- System console
	- Volume Name	- CONSOLE:
	- Attribute	- serial
	- UnitClear	- Clear type-ahead and keyboard buffers.
	- UnitRead	- Echo input character, zero-fill remainder of BUFF instead of returning end-of-file marker.
	- UnitWrite	- Output to system console
	- UnitBusy	- Return FALSE
	- UnitWait	-
	- UnitStatus	- Return keyboard type-ahead information
#2:	- Device	- System console
	- Volume Name	- SYSTERM:
	- Attribute	- Serial
	- UnitClear	- Clear type-ahead and keyboard buffers
	- UnitRead	- Input from system keyboard
	- UnitWrite	- Output to system console
	- UnitBusy	- Return FALSE
	- UnitWait	-
	- UnitStatus	- Return keyboard type-ahead information
#4:	- Device	- Floppy drive 0
	- Volume Name	- user defined
	- Attribute	- Block-structured
	- UnitClear	- Seek to track 0.
	- UnitRead	- Read from floppy
	- UnitWrite	- Write to floppy
	- UnitBusy	- Return FALSE
	- UnitWait	-
	- UnitStatus	- Return disk parameter information

```
#5:  - Device          - Floppy drive 1
     - Volume Name     - user defined
     - Attribute       - Block-structured

     - UnitClear       - Seek to track 0.
     - UnitRead        - Read from floppy
     - UnitWrite       - Write to floppy
     - UnitBusy        - Return FALSE
     - UnitWait        -
     - UnitStatus      - Return disk parameter information


#6:  - Device          - Parallel printer output
     - Volume Name     - PRINTER:
     - Attribute       - Serial

     - UnitClear       -
     - UnitRead        - Bad mode
     - UnitWrite       - Write data to printer
     - UnitBusy        - Return FALSE
     - UnitWait        -
     - UnitStatus      -


#7:  - Device          - Remote input
     - Volume Name     - REMIN:
     - Attribute       - Serial

     - UnitClear       - Clear remote input type-ahead queue
     - UnitRead        - Read data from remote input queue
     - UnitWrite       - Bad mode
     - UnitBusy        - Return FALSE
     - UnitWait        -
     - UnitStatus      - Return remote type-ahead information


#8:  - Device          - Remote output
     - Volume Name     - REMOUT:
     - Attribute       - Serial

     - UnitClear       - Clear remote input type-ahead queue
     - UnitRead        - Bad mode
     - UnitWrite       - Write data to remote port
     - UnitBusy        - Return FALSE
     - UnitWait        - No action
     - UnitStatus      - Return remote type-ahead information
```

```
#9:  - Device          - Optional hard disk virtual floppy 0
     - Volume Name     - user defined
     - Attribute       - Block-structured

     - UnitClear       -
     - UnitRead        - Read from virtual floppy
     - UnitWrite       - Write to virtual floppy
     - UnitBusy        - Return FALSE
     - UnitWait        -
     - UnitStatus      - Return disk parameter information


#10: - Device          - Optional hard disk virtual floppy 1
     - Volume Name     - user defined
     - Attribute       - Block-structured

     - UnitClear       -
     - UnitRead        - Read from virtual floppy
     - UnitWrite       - Write to virtual floppy
     - UnitBusy        - Return FALSE
     - UnitWait        -
     - UnitStatus      - Return disk parameter information


#11: - Device          - Optional hard disk virtual floppy 2
     - Volume Name     - user defined
     - Attribute       - Block-structured

     - UnitClear       -
     - UnitRead        - Read from virtual floppy
     - UnitWrite       - Write to virtual floppy
     - UnitBusy        - Return FALSE
     - UnitWait        -
     - UnitStatus      - Return disk parameter information


#12: - Device          - Optional hard disk virtual floppy 3
     - Volume Name     - user defined
     - Attribute       - Block-structured

     - UnitClear       -
     - UnitRead        - Read from virtual floppy
     - UnitWrite       - Write to virtual floppy
     - UnitBusy        - Return FALSE
     - UnitWait        -
     - UnitStatus      - Return disk parameter information
```

Appendix E

RESERVED WORDS

Standard Pascal Reserved Words

and
array
begin
case
const
div
do
downto
else
end
file
for
function
goto
if
in
label
mod
not
of
or
packed
procedure
program
record
repeat
set
then
to
type
until
var
while
with

NOTE: NIL is a predefined identifier in UCSD Pascal.

UCSD Pascal Reserved Words

external
forward
interface
implementation
process
segment
separate
unit
uses

Appendix F

PREDECLARED IDENTIFIERS

Standard Pascal Predeclared Identifiers

abs
arctan
boolean
char
chr
cos
dispose
eof
eoln
exp
false
get
input
integer
ln
maxint
new
odd
ord
output
page
pred
put
read
readln
real
reset
rewrite
round
sin
sqr
sqrt
succ
text
true
trunc
write
writeln

UCSD Pascal Predeclared Identifiers

atan
attach
blockread
blockwrite
close
concat
copy
delete
exit
fillchar
gotoxy
halt
idsearch
insert
interactive
ioresult
keyboard
length
mark
memavail
memlock
memswap
moveleft
moveright
nil
opennew
openold
pos
processid
pwroften
release
scan
seek
semaphore
seminit
signal
sizeof
start
str
string
time
treesearch
unitbusy
unitclear
unitread
unitstatus
unitwait
unitwrite
varavail
vardispose
varnew
wait

NOTE: NIL is a reserved word in standard Pascal.

Appendix G

IMPLEMENTATION LIMITS

G.1 Quantitative Limits

Maximum number of segments in a program: 255
Maximum number of procedures in a segment: 255
Maximum level of nested procedures: 8
Maximum level of nested statements: 12
Maximum size of a procedure: no limit
Maximum size of variables in a procedure: 32766 words
Maximum size of a record or array: 32766 words
Maximum size of a set: 4080 elements
Maximum size of a string: 255 characters
Integer range: -32768 .. 32767 (no overflow checking)
Long integer accuracy: up to 36 digits
Real range: -3.0E38 .. 3.0E38 (2-word reals, approximate)
    -1.0E308 .. 3.0E308 (4-word reals, approximate)
Real accuracy: up to 6 significant digits (2-word reals)
    up to 14 significant digits (4-word reals)

G.2 Sets

An integer subrange type encompassing negative integer values may not be used as the base type of a set in UCSD Pascal. "Negative" sets compile successfully, but cause execution error 1 ("Value range error") when they are assigned negative values.

Example of an invalid set:

```
program revelation;
var nuclear: set of -66..6;
    solar: set of 3..33;
begin
  solar := [5];
  nuclear := [-30];   { program crashes here }
end.
```

G.3 Mixed Expression Evaluation

The lack of integer overflow checking can affect expressions mixing integers with long integers or reals. The compiler evaluates mixed expressions left-to-right; the expression is evaluated with integer operations until either an operand of the final type (long integer or real) is encountered or the end of the expression is reached. Only at this point does the compiler convert the expression (sub)result to the final type; however, the integer-valued expression may have already overflowed.

Example of mixed expression misevaluation:

```
program mal;
var I: integer;
    R: real;
begin
  I := 20000;
  R := 3.0;
  writeln(I + 20000 + R);
```

In this example, the compiler emits code to perform an integer addition of the integer variable I and the integer constant 20000. The integer result is then converted to type real and added to the real variable to obtain the expression result. Unfortunately, the integer addition overflows, resulting in an incorrect integer subresult; the error is merely propagated by the subsequent real operations.

This problem can be avoided by reordering expressions so that real or long integer operands precede the integer operands; this forces the compiler to convert integer operands to the final type as they are encountered.

G.4 NIL Pointer References

UCSD Pascal does not detect dynamic variable references through pointer variables containing the value NIL (these should be flagged with execution error 7, but are not).

G.5 Record Variant Accesses

UCSD Pascal provides no checks for the detection of invalid record variant references (i.e., accessing a record variant which does not correspond with the tag field value).

G.6 FOR Statements

FOR statements with a final value of MAXINT become infinite loops. Avoid using MAXINT (and -MAXINT) as the initial and final values.

G.7 Special Symbols

Some of the special symbols in UCSD Pascal are internally equivalent; they may be substituted for each other without affecting the compilability of a program.

```
SEGMENT is equivalent to PROGRAM

:  is equivalent to  ..
```

G.8 MOD and DIV with Negative Arguments

The result of a MOD or DIV operation involving negative arguments differs between implementations of UCSD Pascal. The result of a DIV operation with positive arguments is always truncated. When using negative operands, some processors round the result of a DIV towards the larger integer (less negative); some processors round towards the smaller (more negative). Since `a MOD b` is defined to be `a - (a DIV b) * b`, the values returned by MOD are affected by the result of DIV.

Appendix H

COMPILER SYNTAX ERRORS

With thanks to SofTech Microsystems.

1. Error in simple type
2. Identifier expected
3. unimplemented error
4. ’)’ expected
5. ’:’ expected
6. Illegal symbol (terminator expected)
7. Error in parameter list
8. ’OF’ expected
9. ’(’ expected
10. Error in type
11. ’[’ expected
12. ’]’ expected
13. ’END’ expected
14. Semicolon expected
15. Integer expected
16. ’=’ expected
17. ’BEGIN’ expected
18. Error in declaration part
19. Error in <field-list>
20. ’.’ expected
21. ’*’ expected

22. ’INTERFACE’ expected
23. ’IMPLEMENTATION’ expected
24. ’UNIT’ expected
50. Error in constant
51. ’:=’ expected
52. ’THEN’ expected
53. ’UNTIL’ expected
54. ’DO’ expected
55. ’TO’ or ’DOWNTO’ expected in for statement
56. ’IF’ expected
57. ’FILE’ expected
58. Error in <factor> (bad expression)
59. Error in variable
60. Must be of type ’SEMAPHORE’
61. Must be of type ’PROCESSID’
62. Process not allowed at this nesting level
63. Only main task may start processes
101. Identifier declared twice
102. Low bound exceeds high bound
103. Identifier is not of the appropriate class
104. Undeclared identifier
105. Sign not allowed
106. Number expected
107. Incompatible subrange types
108. File not allowed here
109. Type must not be real
110. <tagfield> type must be scalar or subrange
111. Incompatible with <tagfield> part
112. Index type must not be real
113. Index type must be a scalar or a subrange
114. Base type must not be real
115. Base type must be a scalar or a subrange
116. Error in type of standard procedure parameter
117. Unsatisfied forward reference
118. Forward reference type identifier in variable declaration
119. Re-specified params not OK for a forward declared procedure
120. Function result type must be scalar, subrange or pointer
121. File value parameter not allowed
122. A forward declared function’s result type can’t be re-specified
123. Missing result type in function declaration
124. F-format for reals only
125. Error in type of standard procedure parameter

126. Number of parameters does not agree with declaration
127. Illegal parameter substitution
128. Result type does not agree with declaration
129. Type conflict of operands
130. Expression is not of set type
131. Tests on equality allowed only
132. Strict inclusion not allowed
133. File comparison not allowed
134. Illegal type of operand(s)
135. Type of operand must be Boolean
136. Set element type must be scalar or subrange
137. Set element types must be compatible
138. Type of variable is not array
139. Index type is not compatible with the declaration
140. Type of variable is not record
141. Type of variable must be file or pointer
142. Illegal parameter solution
143. Illegal type of loop control variable
144. Illegal type of expression
145. Type conflict
146. Assignment of files not allowed
147. Label type incompatible with selecting expression
148. Subrange bounds must be scalar
149. Index type must be integer
150. Assignment to standard function is not allowed
151. Assignment to formal function is not allowed
152. No such field in this record
153. Type error in read
154. Actual parameter must be a variable
155. Control variable cannot be formal or non-local
156. Multidefined case label
157. Too many cases in case statement
158. No such variant in this record
159. Real or string tagfields not allowed
160. Previous declaration was not forward
161. Again forward declared
162. Parameter size must be constant
163. Missing variant in declaration
164. Substition of standard proc/func not allowed
165. Multidefined label
166. Multideclared label
167. Undeclared label

168. Undefined label
169. Error in base set
170. Value parameter expected
171. Standard file was re-declared
172. Undeclared external file
173. Fortran procedure or function expected
174. Pascal function or procedure expected
175. Semaphore value parameter not allowed
176. Undefined forward procedure
182. Nested units not allowed
183. External declaration not allowed at this level
184. External declaration not allowed in INTERFACE section
185. Segment declaration not allowed in INTERFACE section
186. Labels not allowed in INTERFACE section
187. Attempt to open library unsuccessful
188. Unit not declared in previous USES
189. 'USES' not allowed at this nesting level
190. Unit not in library
191. Forward declaration was not segment
192. Forward declaration was segment
193. Not enough room for this operation
194. Flag must be declared at top of program
195. Unit not importable
201. Error in real number - digit expected
202. String constant must not exceed source line
203. Integer constant exceeds range
204. 8 or 9 in octal number
250. Too many scopes of nested identifiers
251. Too many nested procedures or functions
252. Too many forward references of procedure entries
253. Procedure too long
254. Too many long constants in this procedure
256. Too many external references
257. Too many externals
258. Too many local files
259. Expression too complicated
300. Division by zero
301. No case provided for this value
302. Index expression out of bounds
303. Value to be assigned is out of bounds
304. Element expression out of range
398. Implementation restriction

399. Implementation restriction
400. Illegal character in text
401. Unexpected end of input
402. Error in writing code file, not enough room
403. Error in reading include file
404. Error in writing list file, not enough room
405. 'PROGRAM' or 'UNIT' expected
406. Include file not legal
407. Include file nesting limit exceeded
408. INTERFACE section not contained in one file
409. Unit name reserved for system
410. Disk error
500. Assembler Error

Appendix I

ASCII CHARACTER SET

0	000	00	NUL	32	040	20	SP	64	100	40	@	96	140	60	‘
1	001	01	SOH	33	040	21	!	65	101	41	A	97	141	64	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	78	104	44	D	100	144	64	d
5	005	05	ENQ	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	'	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(	72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29	)	73	111	49	I	105	151	69	i
10	012	0A	LF	42	052	2A	*	74	112	4A	J	106	152	6A	j
11	013	0B	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	0C	FF	44	054	2C	,	76	114	4C	L	108	154	6C	l
13	015	0D	CR	45	055	2D	–	77	115	4D	M	109	155	6D	m
14	016	0E	SO	46	056	2E	.	78	116	4E	N	110	156	6E	n
15	017	0F	SI	47	057	2F	/	79	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	022	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	064	35	5	85	125	55	U	117	165	75	u
22	026	16	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	88	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[	123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C	|
29	035	1D	GS	61	075	3D	=	93	135	5D	]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	↑	126	176	7E	~
31	307	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DEL

Appendix J

DIFFERENCES BETWEEN UCSD VERSIONS

This section describes differences between versions II, III and IV of the UCSD Pascal system. Executable code files are not transportable across versions; however, UCSD Pascal programs are generally source compatible across versions (i.e.,they may be recompiled without changes to run on a different version). Source incompatibilities result from changes in some of the UCSD Pascal extensions; programs written in the UCSD variant of Standard Pascal are completely source compatible across versions.

J.1 Concurrency

Concurrency is not included in Version II UCSD Pascal; it exists only in versions III and IV. Concurrency in versions III and IV is identical.

J.2 NOT

The NOT operator in versions II and IV returns the full-word logical negation of its operand. NOT in Version III (releases H.0 and beyond) returns the boolean negation of its operand (i.e.,the low-order bit of the operand is negated, but the high-order 15 bits of the result are zeroed).

J.3 Long Integers

In Version III, programs containing long integers must use the system unit named LONGINT. This is unnecessary in versions II and IV.

J.4 Transcendental Functions

In some Version II systems, programs calling the transcendental functions must use the system unit named TRANSCEND. This is not necessary in versions III and IV.

J.5 Segments

Some implementations of Version II allow a program to contain up to 8 segments. Version IV allows up to 255 segments.

J.6 Units

Separate units are unique to Version II. Data units (i.e.,units containing only an interface section of types and variables) are allowed in versions II.1 and III.1. Version IV units may contain segment procedures, files, and termination sections. Versions IV, II.1, and some releases of Version II allow intialization sections.

J.7 TREESEARCH

The ordering of trees built by TREESEARCH is implementation dependent and varies across machines. TREESEARCH itself works correctly on all systems; only manual tree traversals are affected by this property.

J.8 Intrinsics

The intrinsics PMACHINE, ATTACH, SEMINIT, SIGNAL, START, and WAIT are present in versions III and IV. Version IV allow the memory management intrinsics MEMLOCK, MEMSWAP, DISPOSE, VARNEW, VARDISPOSE, and VARAVAIL. Note that the file intrinsics OPENOLD and OPENNEW are not present in Version IV. Version III allows the memory management intrinsic RMEMAVAIL. Versions IV and II.1 allow the Unit I/O UNITSTATUS intrinsic. The I/O redirection intrinsic

REDIRECT is provided only in Version IV.

The value of the UNITBUSY boolean function in Version III systems distributed by Western Digital differs with values returned on other UCSD Pascal systems. The UNITBUSY function value should be negated when transferring software between Western Digital Version III systems and others.

J.9 I/O Redirection

I/O redirection is provided only in Version IV.

J.10 Pointer Comparison

Extended pointer comparison exists in versions III and IV.

J.11 Procedure Size

The restrictions on procedure size are greatly relaxed in Version IV. This can affect the transportability of programs developed on Version IV.

Bibliography

Introductory Pascal Texts:

Findlay, W., and Watt, D.A. 1978 *Pascal: An Introduction to Methodical Programming.* Computer Science Press, Potomac, MD

Wilson, I.R., and Addyman, A.M. 1978. *A Practical Introduction to Pascal.* Springer-Verlag, New York, NY

Cooper, D., and Clancy, M. 1982. *Oh! Pascal!* W.W. Norton & Company, New York, NY

Welsh, J., and Elder, J. 1979. *Introduction to Pascal.* Prentice-Hall International, New York, NY

Introductory p-System Texts:

Bowles, K.B. 1980. *Beginner's Guide for the UCSD Pascal System.* Byte/McGraw-Hill, New York, NY

Overgaard, M., and Stringfellow, S. 1983. *Personal Computing with the UCSD p-System.* Prentice-Hall, Englewood Cliffs, NJ

Luehrmann, A., and Peckham, H. 1981. *Apple Pascal: A Hands-On Approach.* McGraw-Hill, New York, NY

Grant, W.G., and Butah, J. 1982. *Introduction to the UCSD p-System.* Sybex, Berkeley, CA

Pascal Language Reference Texts:

Jensen, K., and Wirth, N. 1974. *Pascal User Manual and Report.* Springer-Verlag, New York, NY

IEEE Pascal Standards Committee, 1983. *American National Standard Pascal Computer Programming Language.* The Institute of Electrical and Electronic Engineers, Inc., New York, NY

Ledgard, H. 1984. *The American Pascal Standard with Annotations.* Springer-Verlag, New York, NY

Tiberghien, J. 1981. *The Pascal Handbook.* Sybex,Berkeley, CA

More Advanced Programming Using Pascal:

Tenenbaum, A.M., and Augenstein, M.J. 1981. *Data Structures Using Pascal.* Prentice-Hall, Englewood Cliffs, NJ

Wirth, N. 1976. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, NJ

Schneider, M.S., and Bruell, S.C. 1981. *Advanced Programming and Problem Solving with Pascal./uf John Wiley & Sons, New York, NY*

Glinert, E. 1983. Introduction to Computer Science Using Pascal. Prentice-Hall International, New York, NY

Kernighan, B.W., and Plauger, P.J. 1981. *Software Tools in Pascal.* Addison-Wesley, Reading, MA

Sand, P.A. 1984. *Advanced Pascal Programming Techniques.* Osborne/McGraw-Hill, Berkeley, CA

More Advanced p-System Texts:

Clark, R. and Koehler, S. 1982. *The UCSD Pascal Handbook.* Prentice-Hall, Englewood Cliffs, NJ

Barron, D.W., Editor, 1981. *Pascal: The Language and its Implementation.* John Wiley & Sons, New York, NY

Pemberton, S. and Daniels, M.C. 1982. *Pascal Implementation: The P4 Compiler.* John Wiley & Sons, New York, NY

Hyde, R. 1983. *p-Source.* Datamost, Chatsworth, CA

De Groat, R., Editor, 1982. *All About Pascal.* Apple PugetSound Program Library Exchange,Renton, WA

Applications in Pascal:

Swan, T. 1983. *Pascal Programs for Business.* Hayden, Hasbrouck Heights, NJ

Swan, T. 1984. *Pascal Programs for Data Base Management.* Hayden, Hasbrouck Heights, NJ

Cooper, J.W. 1981. *Introduction to Pascal for Scientists.* John Wiley & Sons, New York, NY

Pyster, A.B. 1980. *Compiler Design and Construction.* Van Nostrand Reinhold, New York, NY

Vile, R.C. Jr. 1984. *Programming Your Own Adventure Games in Pascal.* Tab Books, Blue Ridge Summit, PA

Davidson, G., and Poole, L., Editors, 1982. *Practical Pascal Program.* Osborne/McGraw-Hill, Berkeley, CA

Davidson, G., Poole, L., and Borchers, M., Editors, 1982. *Some Common Pascal Programs.* Osborne/McGraw-Hill, Berkeley, CA

Index

- * -

*** 48, 54

- 6 -

68000 . . . . 85, 296

- 8 -

8080 3
8088 . . . . 85, 214

- A -

Activation Record 39
ADDR Function 197, 214
AND 186
Apple Pascal 2, 3, 39, 44, 49, 54, 149, 152, 154, 218, 225, 256, 289
Arbitrary Access
  of bit fields 179
  of bits 180
  of words 178
ARCTAN 98
Array 241
  indices of 226
Array Indexing
  Using BOOLEAN or CHAR
    Subscripts 10
Arrays
  Type Compatibility 16
Assembler 319
Assembler Language 38, 198, 223
Associate Time 292
ATAN 98, 113
ATTACH 35, 36, 114

- B -

.BACK 255
Backus-Naur Form 4
.BAD 255
Binary Semaphore 31, 32
Binary Tree 103, 135
32-bit Integer 96, 230
Block 5, 188, 247, 254, 258
Block File 22, 55, 60
Block Number 5, 247
BLOCKREAD 55, 60, 114, 202, 283, 320
Block-structured Device 245, 247
Block-structured Unit 245, 247
Block-structured Volume 247, 249
BLOCKWRITE 55, 60, 115, 267, 283
BNF 4
BOOLEAN
    Arithmetic with 11
    Complementing 10
Boundary Restriction 176
Break Key 222
Byte Array Manipulation 79
Byte Magazine 195
Byte Sex 161, 296
Bytes-in-last-block 252, 254

- C -

CASE 7, 238
CHAIN 116, 130, 223, 319
Character Prompt 204
CHR 10, 186, 238
Clock Access 95
Clock Handler 37
CLOSE 14, 54, 56, 117, 131
.CODE 255
Code File 253
Code File Structure 293
Code Offset 146, 215, 232
Code Optimization 226
Code Pool 39, 302, 305
    External 40
    Management
        Strategy 40, 42
Code Segment 38, 39, 51, 53, 215
COMMANDIO 116, 119, 129, 277
Comments 15
Compilation Unit 291
Compiled Listings 145
Compile Flags 150
Compile Option 142
Compiler 256, 346
Compiler Error 321
Compiler Interface 319
Compiler Listing 215, 257
Compile-time Functions 79
CONCAT 63, 65, 118, 236
Concurrency 23
Conditional
        Compilation 150, 217
Conformant Arrays 186
Constant Pool 298
Cooperating Processes 33
COPY 63, 65, 118, 165
COPYDUPDIR 251
Copyright Notice 153
Counting Semaphore 31
CP/M 170
Critical Section 32, 188
CRUNCH 56, 117
Current Task 23

- D -

Data File 253, 254
Data Prompt 203
DATASIZE 299
Data Unit 49
DECODE 28, 253, 271, 311
DELETE 63, 65, 118, 165, 237
Device Access 212
Device Driver 210
Device I/O 84
DIRINFO Unit 220
Disk Directory 249, 251, 282
Disk Drive 246
Disk File 249, 251, 252
Disk Unit 245, 249
Disk Volume 249
DISPOSE 9, 26, 72, 102, 195, 310
Dummy Segment 157
Duplicate Directory 251, 288
Dynamic Arrays 186
Dynamic Variable
Management 68

- E -

Editor 254
EOF 12, 59, 62, 132
EOLN 59
EREC 292, 301
Error Handler 152
ERRORS 257
Event 35
EXCEPTION 119
Executable Units 162
Execution Error 214
Execution State 23
EXIT 8, 98, 119, 121
  In Units 48
Exit Code 312
EXITIC 300
Extended Comparisons 77
Extended Precision
Arithmetic 74
EXTERNAL 295
External Code Pool 40
External File 130, 131
External Procedure 192, 198, 241

- F -

FIB 306, 310, 320
File 54
File Attributes 252
File Date 252, 254
File Designator 245, 259
FILEID 113
File Identifier 245, 249, 259
File Length 252, 254
File Name 5, 245, 252
FILEOPS 309
File Prompt 205
Filer 259
File Suffix 252, 254, 259, 262, 263
File System 99, 244
File Title 252, 255, 257, 259, 262
File Type 252, 253
File Variables 15, 310
FILLCHAR 81, 120, 133, 187, 189
FOR 109
FORWARD 9
Forward Declaration 41
Fraction Length 13
Full-word Logical
Operations 186
Functional Parameters 14

Functions 192

- G -

Garbage Collection 195
General Prompt 263
GET 36, 56, 58, 60, 61, 130, 132, 165
GETCVAL 225
GOTO 8, 99, 153, 239
GOTO Statements
  In Units 48
GOTOXY 94, 121, 165, 203, 228

- H -

HALT 103, 121
Hard Disk 247
Heap 26, 28, 39, 69, 102, 187, 195, 302, 305

- I -

IBM PC 229
$I Compile Option 60
Identifiers 92
IDSEARCH 106, 122
IMPLEMENTATION 45, 313
Implementation Section 45
Include File 92, 147
Initialization Code 162, 313
Initialization Section 45
Inline Machine Code 89
INPUT 12, 14, 59
Input Prompt 262
INSERT 63, 65, 76, 118, 122, 165, 237
Integer Overflow 186
Integer Prompt 204
INTERACTIVE 54, 58
Interactive File 58
INTERFACE 45, 49, 294, 309
Interface Section 45
Interpreter 257
Inter-Program Parameter
  Passing 223
Interrupt 36
  Hardware Generated 35
Interrupt Handling 24, 35
Intrinsic 22, 112
Intrinsic Unit 54
I/O Check 114, 115, 151
I/O Completion Status 100
I/O Device 245, 246
I/O Operation
  Effect on Tasks 38
I/O Redirection 116, 354
IORESULT 60, 84, 87, 100, 122, 203

- K -

KERNEL 159, 280, 306
KEYBOARD 54, 59
Keyboard 246
Keyboard File 204
K(runch 258

- L -

Labels
  In Units 48
LENGTH 63, 65, 123
Length Specifier 252, 258, 259
Lexical Level 156
Library 52, 148, 158, 256
Library Modules 22
Linked List
  of files 310
Linker 200

LOCK 56, 117
Logical Volume 247
Long Integer 74, 230, 241
    Parameter 76
    Reading from
        Console 76

- M -

Main Task 25, 30
MARK 9, 26, 28, 69, 123, 130
MARKDUPDIR 251
Master File Update 307
MAXINT 19, 315
MEMAVAIL 102, 124, 186, 188
MEMLOCK 40, 42, 72, 124, 125, 155, 303
Memory Management 38, 39, 51, 305
Memory-Mapped I/O 212
MEMSWAP 40, 42, 124, 125, 155, 303
Meta-words 4
MOD 152
Monitor 278
MOVELEFT 81, 125, 133, 187, 202, 224, 320
MOVERIGHT 81, 126, 133, 187
Multiprocessing 102
Multiword Constants 227
Mutual Exclusion 32

- N -

Name Compatibility 16
Native Code 159, 221
Native Code Generator 38
Nesting Level 146
NEW 69, 123, 174, 187, 189, 310
NIL 9, 71, 114
    Attaching to
        Semaphore 35
Noload 154
NORMAL 56, 117
NOT 10, 186, 352

- O -

ODD 10, 186
Offline 246
Online 246
OPCODES 257
OPENNEW 56, 127, 353
OPENOLD 56, 127, 353
Operating System 253
Operating System
        Procedures 309
OR 186
ORD 94, 186, 201, 238
OTHERWISE Clause 8
OUTPUT 14
Output Prompt 263

- P -

PACK 13
Packed Array 171
Packed Field References 228
Packed Record 173
Packed Variables 170
Packing Rules 174
PAGE 165
Page Break
    in compiler listing 147
Partial Boolean
        Expression 83, 131
Pascal User Manual and
        Report 26, 47
PATCH 253
PDP-11 . . . . 2, 41, 85, 88, 211, 296

Peeking 179
Physical Unit 245
Physical Unit Number 245
P__MACHINE 22, 89, 127,
        200, 214, 308
Pointer 18, 179, 193
    comparison 94
    type conversion 94
POS 63, 65, 128, 165
Powers of Ten 97
PRED 165
P(refix 250
Prefixed Volume 248, 250,
        259
Prefix Options 250
Primary Segment 162
Printer 246
Print Spooler 36
Priority 24, 29
Private Semaphore 33
Procedural Parameters 14
Procedure Call 240
Procedure Dictionary 297
Procedure Number 146, 215,
        216, 313
Procedure Parameters 241
Procedure Termination 98
Process 24, 26, 98
PROCESSID 25, 27
Programming Practices 169
Program Segmentation 38
Prompt Conventions 262
Pseudo-comment 142
PURGE 56, 117
PUT 12, 61, 132, 165
PWROFTEN 97, 128

- Q -

Q-Bus 211
QUICKSTART 301
Quiet Console 154

- R -

Random Access File 61
Range Check 152, 227
Range Checking
    with Block I/O 60
$R Compile Option 64
READ 13, 36, 58, 63, 65, 75,
        210
READLN 13, 58, 63, 65, 165,
        253
Ready Queue 23, 29, 31
Ready-To-Run Task 23
Real 75, 230
Real Constant 231, 298
REALCONV 222, 231, 299
REALOPS Unit 161
Real Prompt 206
Real Size 160, 299
Record 241
Records
    Type Compatibility 17
REDIRECT 129, 130, 353
Relational Operators 77
    On packed data 78
    On pointers 79
    On strings 78
RELEASE 9, 26, 28, 69, 123,
        129
Relocation List 297
Reserved Word 21, 340
RESET 12, 14, 54, 55, 58, 62,
        117, 127, 130, 206, 283
Resident 154
Reverse Field Allocation 177
REWRITE 12, 14, 54, 55, 62,
        127, 131, 206
Root Volume 250

- S -

SCAN 22, 83, 131
Screen Control 94
SCREENOPS 270
SEEK 55, 61, 132, 165
Segment 39, 41, 156, 288
Segment Dictionary 291, 293, 294
Segment Format 296
Segment Number 291
Segment Procedure 191
Segment Reference List 292, 294, 295, 300
Segment Residency 154
Selective USES 50
Semaphore 24, 30, 214
SEMINIT 31, 133
Separate Compilation 44
Serial Device 245, 247, 253
Serial Unit 245, 247
Serial Volume 247, 257
Set 18, 180, 228
    Type Compatibility 16
Set Constant 228
SETCVAL 225
SETUP 256, 270
Short Form p-codes 231
SIB 292, 301
SIGNAL 31, 32, 133
SIZEOF 79, 120, 133, 171, 315
Sorting 201
Sort/Merge Utility 310
SQR 165
Stack 39, 305, 306
Stack Overflow 28, 39, 69
Stack Size 24
Stack Space 28
Standard Input 12, 54, 246
Standard Output 12, 55, 246
START 25, 134
Starting Block 252, 254
Storage Allocation 170, 174
STR 76, 134, 165
String 63, 223, 241
String Constant 172
String Manipulation 236
STRINGOPS 309
String Option 142
String Overflow 64
String Parameter 66
Structure Compatibility 16
Structured Field 174
Subranges 174
Subsidiary Task 25
Subsidiary Volume 246, 260
SUCC 165
Suspended Task 24
.SVOL 255
Swapping 149
Switch Option 142
Symbolic Debugging 161
Synchronization 33
Synchronous I/O 30
Syntax Error 346
System Clock 37
SYSTEM.COMPILER 256
System Constants 281
System Data Structures 266
System Date 281
System File Title 255, 257
System Globals 158, 280
SYSTEM.LIBRARY 53, 256
System Library 52
SYSTEM.MENU 256
SYSTEM.MISCINFO 255, 256, 270
SYSTEM.PASCAL 256
System Programs 155
System Shell 354
SYSTEM.SPOOLER 257

SYSTEM.STARTUP 256
SYSTEM.SWAPDISK 148, 256
SYSTEM.SYNTAX 256
System Units 266
System Volume 248, 250
SYSTEM.WRK.CODE 256
SYSTEM.WRK.TEXT 256

- T -

Task 23
Task Blockage 37
Task Identifier 24, 27, 134
Task Priority 134
Task Scheduling Policy 29
Task Stack 134
Task Switch 23, 29
Task Synchronization 24, 32
Termination Code 162
Termination Section 45
.TEXT 255
TextFile 57, 247, 253, 320
TextFile Format 267
TIME 95, 135, 186
Time Slicing 24, 37
Tiny Compiler 165
Title
  in compiler listing 147
TREESEARCH 103, 135, 353
TRUNC 75, 76
Type-ahead Buffer 246
Type-checking Prompt 262
Type Compatibility 16, 315
Type Rules 196

- U -

$U Compiler Option 53
UCSD Pascal Users' Society 158, 271, 280
UNIT 45, 99, 148, 309
Unit 5, 113, 150, 162, 186, 192, 217, 290
UNITBUSY 87, 136, 332
UNITCLEAR 87, 136, 203, 332
Unit I/O 261
Unit Linkage 52
Unit Number 5, 246, 248, 259
UNITREAD 36, 84, 136, 165, 332
UNITSTATUS 38, 88, 137, 332, 336
UNITWAIT 87, 138, 332
UNITWRITE 84, 138, 267, 332
UNIV 113
UNPACK 13
Unsigned Integer 72, 182
Untyped Parameters 196
User-Defined Serial Device 246
User File Title 257
User Library 52, 218
USERLIB.TEXT 53, 149, 257
USES 45, 50, 164

- V -

Value Parameter 242
  String 67
VARAVAIL 72, 102, 139, 190, 201
VARDISPOSE 72, 139, 191
Variable Parameter
  String 67
Variable Scope 232
Variable-sized Array Allocation 186
Variant Record 174, 179, 201

Variant Records 14
VARNEW 72, 140, 186, 189, 201
VAR Parameter 171, 195
Volume Identifier 245, 248, 259
Volume Name 247, 248, 259

- W -

WAIT 31, 32, 133, 140
Wait Queue 24, 30
Western Digital
    MicroEngine 3, 36
WILD Unit 220
Window Variable 59, 307
WITH 234
Workfile 256, 319
WRITE 13, 36, 63, 75
WRITELN 13, 63, 217

- Z -

Z80 .... 85, 198

Biography

Eliakim Willner, CDP is a consultant specializing in mini and microcomputer-based systems. Mr. Willner has designed and coded many applications in UCSD Pascal, which is his language of preference. He heads a group which is currently maintaining and marketing the UCSD Pascal System for Digital Equipment Corporation PDP-11 computers.

Mr. Willner is an adjunct lecturer of data processing at Kingsborough Community College of the City University of New York. He has published articles on data processing in *Byte*, *Small Systems World* and other publications. Mr. Willner is a member of the ACM and the IEEE Computer Society and is Chairman of the Board of Directors of the UCSD Pascal Users' Society.

Barry Demchak began programming in Pascal at the University of California, San Diego while working on the UCSD Pascal Project. Later, he maintained the UCSD operating system and helped develop the p-System I/O system. He graduated from UCSD in 1979 with a bachelor's degree in Computer Science with a minor in Economics.

Barry worked at SofTech Microsystems where he helped design and implement the p-System Version IV p-machine and operating system. He is currently a consultant with Software Construction, Inc. in La Jolla, California, where he is still actively involved with the p-System. He enjoys skiing, movies and the theater.

ADVANCED UCSD PASCAL PROGRAMMING TECHNIQUES

Eliakim Willner • Barry Demchak

In his Preface to this absorbing new book, Ken Bowles writes, "This is a book for 'serious' users of the UCSD Pascal System as most widely distributed on many different personal computers. It is a book that I wish had been available five years ago.

"Niklaus Wirth designed Pascal to be a teaching language, one that would be an expression of the systematic or 'structured' methods of writing better programs. Like many others, we at UCSD found Pascal to be too good to be limited to teaching only. We saw it as a superb vehicle for creating large, complex system programs—not just the USCD p-System itself, but a wide variety of large and complex application programs and products. But to use Wirth's Pascal for these applications in practice on a microcomputer required subtle changes and extensions in Wirth's teaching language Pascal. Thus was born the UCSD Pascal language.

"In writing this book, **Barry Demchak** and **Eli Willner** have provided a highly readable compendium of the essential lore needed by any serious user of the USCD p-System. As both a collection of suggested techniques and a reference work, a short study of this book will often save large amounts of time for both professional programmers and advanced students of computer science who use the USCD p-System."

0 21898 01161 9

PRENTICE-HALL, INC.
Englewood Cliffs, N.J. 07632

ISBN 0-13-011610-6